Register Allocation

Martin Kellogg

Course Announcements

- The PA4 leaderboard is still Coming Soon™

Course Announcements

- The PA4 leaderboard is still Coming Soon™
  - Unfortunately the technical problems here were tougher than I expected, so I am changing how it works
    - More details to come in the next day or two…

Course Announcements

- The PA4 leaderboard is still Coming Soon™
  - Unfortunately the technical problems here were tougher than I expected, so I am changing how it works
    - More details to come in the next day or two…
- Due to popular demand, I have **changed** PA4c1's specification to also allow the input to be a `.cl-type` file (originally was TAC)
  - PA4c1 is due April 28, and is mostly optional

Register Allocation

Register Allocation

- IR code typically assumes **infinitely-many registers** are available
  - but real machines only have a small number of registers :(

Register Allocation

- IR code typically assumes **infinitely-many registers** are available
  - but real machines only have a small number of registers :(
- Task of the ***register allocator***: create a mapping from IR's abstract registers to physical registers

Register Allocation

- IR code typically assumes **infinitely-many registers** are available
  - but real machines only have a small number of registers :(
- Task of the ***register allocator***: create a mapping from IR's abstract registers to physical registers
  - Or, if that's not possible, to memory locations
    - this happens when there aren't enough physical registers

Register Allocation

- IR code typically assumes **infinitely-many registers** are available
  - but real machines only have a small number of registers :(
- Task of the ***register allocator***: create a mapping from IR's abstract registers to physical registers
  - Or, if that's not possible, to memory locations
    - this happens when there aren't enough physical registers
  - Insert code to move values between registers and memory if needed ("***spill code***")

Register Allocation

- IR code typically assumes **infinitely-many registers** are available
  - but real machines only have a small number of registers :(
- Task of the ***register allocator***: create a mapping from IR's abstract registers to physical registers
  - Or, if that's not possible, to memory locations
    - this happens when there aren't enough physical registers
  - Insert code to move values between registers and memory if needed ("***spill code***")
    - Typically we will re-run the instruction scheduler if we ever have to spill a register

Register Allocation: PA3 Version

- Even your PA3 implementation should probably have a simple “register allocator” somewhere

Register Allocation: PA3 Version

- Even your PA3 implementation should probably have a simple "register allocator" somewhere
  - my suggestion: "**everything goes in memory**"

Register Allocation: PA3 Version

- Even your PA3 implementation should probably have a simple "register allocator" somewhere
  - my suggestion: "**everything goes in memory**"
  - then you can insert load code before any operation that requires its operands in registers without worrying about what's in those registers when you do

Register Allocation: PA3 Version

- Even your PA3 implementation should probably have a simple "register allocator" somewhere
  - my suggestion: "**everything goes in memory**"
  - then you can insert load code before any operation that requires its operands in registers without worrying about what's in those registers when you do
- However, one of the **first optimizations** that I suggest you implement for PA4 is a simple register allocator

Register Allocation: PA3 Version

- Even your PA3 implementation should probably have a simple "register allocator" somewhere
  - my suggestion: "**everything goes in memory**"
  - then you can insert load code before any operation that requires its operands in registers without worrying about what's in those registers when you do
- However, one of the **first optimizations** that I suggest you implement for PA4 is a simple register allocator
  - avoiding memory operations is **extremely profitable** 🤑🤑🤑

Register Allocation: PA3 Version

- Even your PA3 implementation should probably have a simple "register allocator" somewhere
  - my suggestion: "**everything goes in memory**"
  - then you can insert load code before any operation that requires its operands in registers without worrying about what's in those registers when you do
- However, one of the **first optimizations** that I suggest you implement for PA4 is a simple register allocator
  - avoiding memory operations is **extremely profitable** 🤑🤑🤑
  - I recommend waiting until later in PA4 to try more complex schemes

Register Allocation: Overview

Register Allocation: Overview

- Register allocation: correctness or optimization
  - depends on **memory model**

Register Allocation: Overview

- Register allocation: correctness or optimization
  - depends on **memory model**
- Local register allocation
  - two approaches: top-down and bottom-up

Register Allocation: Overview

- Register allocation: correctness or optimization
  - depends on **memory model**
- Local register allocation
  - two approaches: top-down and bottom-up
- Regional register allocation + complications
  - **liveness analysis** strikes again!

Register Allocation: Overview

- Register allocation: correctness or optimization
  - depends on **memory model**
- Local register allocation
  - two approaches: top-down and bottom-up
- Regional register allocation + complications
  - **liveness analysis** strikes again!
- Global register allocation via **reduction to graph coloring**
  - including a **union-find** algorithm for fun

Register Allocation: Optimization?

Register Allocation: Optimization?

- Whether register allocation is an optimization depends on the compiler's **memory model**

Register Allocation: Optimization?

- Whether register allocation is an optimization depends on the compiler's **memory model**
- There are two common memory models:

Register Allocation: Optimization?

- Whether register allocation is an optimization depends on the compiler's **memory model**
- There are two common memory models:
  - in a ***register-to-register*** model, every value that could be kept in a register is kept in a register in all IRs
    - values are stored in memory only when the semantics require it (e.g., when an address is passed as a parameter)

Register Allocation: Optimization?

- Whether register allocation is an optimization depends on the compiler's **memory model**
- There are two common memory models:
  - in a ***register-to-register*** model, every value that could be kept in a register is kept in a register in all IRs
    - values are stored in memory only when the semantics require it (e.g., when an address is passed as a parameter)
  - in a ***memory-to-memory*** model, every value is presumed to be stored in memory
    - values are loaded into registers as needed for operations; results are stored back to memory

Register Allocation: O

In one of these models, a register allocator is required **for correctness**; in the other, it's an optimization. Which is which?

- Whether register allocatio
  compiler's **memory model**
- There are two common me
  - in a ***register-to-register*** model, every value that could be kept in a register is kept in a register in all IRs
    - values are stored in memory only when the semantics require it (e.g., when an address is passed as a parameter)
  - in a ***memory-to-memory*** model, every value is presumed to be stored in memory
    - values are loaded into registers as needed for operations; results are stored back to memory

Register Allocation: Optimization?

- In a register-to-register model, the register allocator is required **for correctness**

Register Allocation: Optimization?

- In a register-to-register model, the register allocator is required **for correctness**
  - Code assumes infinite registers; machine physically only has a finite number. Compiler must move some values to memory or it won't have somewhere to store them.

Register Allocation: Optimization?

- In a register-to-register model, the register allocator is required **for correctness**
  - Code assumes infinite registers; machine physically only has a finite number. Compiler must move some values to memory or it won't have somewhere to store them.
- By contrast, in a memory-to-memory model, register allocation is an **optimization**

Register Allocation: Optimization?

- In a register-to-register model, the register allocator is required **for correctness**
  - Code assumes infinite registers; machine physically only has a finite number. Compiler must move some values to memory or it won't have somewhere to store them.
- By contrast, in a memory-to-memory model, register allocation is an **optimization**
  - Memory is slow, but registers are fast. Register allocation converts as many memory ops to register ops as possible.

Register Allocation: Optimization?

- In a register-to-register model, the register allocator is required **for correctness**
  - Code assumes infinite registers; machine physically only has a finite number. Compiler must move some values to memory or it won't have somewhere to store them.
- By contrast, in a memory-to-memory model, register allocation is an **optimization**
  - Memory is slow, but registers are fast. Register allocation converts as many memory ops to register ops as possible.
  - Hopefully, this is where you are after PA3. I will assume this memory model for the rest of this lecture.

Local Register Allocation

- Just like other optimizations, register allocation is simpler at the scale of a **single basic block**

Local Register Allocation

- Just like other optimizations, register allocation is simpler at the scale of a **single basic block**
  - so, let's consider ***local register allocation*** first

Local Register Allocation

- Just like other optimizations, register allocation is simpler at the scale of a **single basic block**
  - so, let's consider ***local register allocation*** first

In this discussion of local allocation, I'm going to assume that all values must be stored in memory **between** basic blocks

Local Register Allocation

- Just like other optimizations, register allocation is simpler at the scale of a **single basic block**
  - so, let's consider ***local register allocation*** first
- Even at the basic block level, register allocation is **NP-complete**, if any one of the following complications exist:

Local Register Allocation

- Just like other optimizations, register allocation is simpler at the scale of a **single basic block**
  - so, let's consider ***local register allocation*** first
- Even at the basic block level, register allocation is **NP-complete**, if any one of the following complications exist:
  - more than one size of data item
    - e.g., a double-width register

Local Register Allocation

- Just like other optimizations, register allocation is simpler at the scale of a **single basic block**
  - so, let's consider ***local register allocation*** first
- Even at the basic block level, register allocation is **NP-complete**, if any one of the following complications exist:
  - more than one size of data item
    - e.g., a double-width register
  - non-uniform memory access costs

Local Register Allocation

- Just like other optimizations, register allocation is simpler at the scale of a **single basic block**
  - so, let's consider ***local register allocation*** first
- Even at the basic block level, register allocation is **NP-complete**, if any one of the following complications exist:
  - more than one size of data item
    - e.g., a double-width register
  - non-uniform memory access costs
  - some values do not need to be stored to memory at the end of their lifetimes (e.g., constants)

Local Register Allocation

- Just like other optimizations, register allocation is simpler at the scale of a **single basic block**
  - so, let's consider ***local register allocation*** first
- Even at the basic block level, register allocation is **NP-complete**, if any one of the following complications exist:
  - more than one size of data item
    - e.g., a double-width register
  - non-uniform memory access costs
  - some values do not need to be stored to memory at the end of their lifetimes (e.g., constants)
  - any control flow (i.e., more than one basic block)

Review (?): NP-Completeness

- Informally, a decision problem is said to ***NP-complete*** if:

Review (?): NP-Completeness

- Informally, a decision problem is said to ***NP-complete*** if:
  - it is “fast” (polynomial time) to **check** if a solution is correct

Review (?): NP-Completeness

- Informally, a decision problem is said to ***NP-complete*** if:
  - it is “fast” (polynomial time) to **check** if a solution is correct
  - it is “slow” (exponential+ time) to **find** a solution

Review (?): NP-Completeness

- Informally, a decision problem is said to ***NP-complete*** if:
  - it is “fast” (polynomial time) to **check** if a solution is correct
  - it is “slow” (exponential+ time) to **find** a solution
  - there exist ***reductions*** both to and from the problem to some other problem that is known to be NP-complete
    - typically boolean satisfiability

Review (?): NP-Completeness

- Informally, a decision problem is said to ***NP-complete*** if:
  - it is "fast" (polynomial time) to **check** if a solution is correct
  - it is "slow" (exponential+ time) to **find** a solution
  - there exist ***reductions*** both to and from the problem to some other problem that is known to be NP-complete
    - typically boolean satisfiability

Recall that a **reduction** is a proof that we could use an oracle for one problem to solve another, so for a problem to be NP-complete we need to prove both that if we can solve it, we could solve SAT and vice-versa

Review (?): NP-Completeness

- Informally, a decision problem is said to ***NP-complete*** if:
  - it is “fast” (polynomial time) to **check** if a solution is correct
  - it is “slow” (exponential+ time) to **find** a solution
  - there exist ***reductions*** both to and from the problem to some other problem that is known to be NP-complete
    - typically boolean satisfiability
- “NP-complete” and “undecidable” are both computational classes
  - Which is harder?

Review (?): NP-Completeness

- Informally, a decision problem is said to ***NP-complete*** if:
  - it is “fast” (polynomial time) to **check** if a solution is correct
  - it is “slow” (exponential+ time) to **find** a solution
  - there exist ***reductions*** both to and from the problem to some other problem that is known to be NP-complete
    - typically boolean satisfiability
- “NP-complete” and “undecidable” are both computational classes
  - Which is harder?
    - undecidable: all undecidable problems are (probably) NP-hard, but not all NP-hard problems are undecidable

Review (?): NP-Completeness

- Informally, a decision problem is said to ***NP-complete*** if:
  - it is “fast” (polynomial time) to **check** if a solution is correct
  - it is “slow” (exponential+ time) to **find** a solution
  - there exist ***reductions*** both to and from the problem to some other problem that is known to be NP-complete
    - typically boolean satisfiability
- “NP-complete” and “undecidable” are both computational classes
  - Which is harder?
    - undecidable: all undecidable problems are (probably) NP-hard, but not all NP-hard problems are undecidable
  - “undecidable” = no **finite-time** algorithm exists

Review (?): NP-Completeness

- Informally, a decision problem is said to ***NP-complete*** if:
  - it is "fast" (polynomial time) to **check** if a solution is correct
  - it is "slow" (exponential+ time) to **find** a solution
  - there exist ***reducti***
    other problem tha
    - typically boole

In practice, for both NP-complete and undecidable problems we typically use **approximate** algorithms ("heuristics")

- "NP-complete" and "undecidable" are both computational classes
  - Which is harder?
    - undecidable: all undecidable problems are (probably) NP-hard, but not all NP-hard problems are undecidable
  - "undecidable" = no **finite-time** algorithm exists

Local Register Allocation: Heuristics

Local Register Allocation: Heuristics

- There isn’t a consensus “best” local allocation heuristic, as there is for e.g., instruction scheduling (list scheduling)

Local Register Allocation: Heuristics

- There isn't a consensus "best" local allocation heuristic, as there is for e.g., instruction scheduling (list scheduling)
- Instead, we will look at two common heuristics:

Local Register Allocation: Heuristics

- There isn't a consensus "best" local allocation heuristic, as there is for e.g., instruction scheduling (list scheduling)
- Instead, we will look at two common heuristics:
  - a ***frequency count*** approach (called "top-down" in the book)

Local Register Allocation: Heuristics

- There isn't a consensus "best" local allocation heuristic, as there is for e.g., instruction scheduling (list scheduling)
- Instead, we will look at two common heuristics:
  - a ***frequency count*** approach (called "top-down" in the book)
  - a ***greedy*** allocator (called "bottom-up" in the book)

Local Register Allocation: Heuristics

- There isn't a consensus "best" local allocation heuristic, as there is for e.g., instruction scheduling (list scheduling)
- Instead, we will look at two common heuristics:
  - a ***frequency count*** approach (called "top-down" in the book)
  - a ***greedy*** allocator (called "bottom-up" in the book)
- For both cases, we will assume that the input is a basic block that uses some number of virtual registers ***v***

Local Register Allocation: Heuristics

- There isn't a consensus "best" local allocation heuristic, as there is for e.g., instruction scheduling (list scheduling)
- Instead, we will look at two common heuristics:
  - a ***frequency count*** approach (called "top-down" in the book)
  - a ***greedy*** allocator (called "bottom-up" in the book)
- For both cases, we will assume that the input is a basic block that uses some number of virtual registers ***v***
  - our goal: create a semantically-equivalent block that uses some fixed number of physical registers ***k***

Local Register Allocation: Heuristics

- There isn't a consensus "best" local allocation heuristic, as there is for e.g., instruction scheduling (list scheduling)
- Instead, we will look at two common heuristics:
  - a ***frequency count*** approach (called "top-down" in the book)
  - a ***greedy*** allocator (called "bottom-up" in the book)
- For both cases, we will assume that the input is a basic block that uses some number of virtual registers ***v***
  - our goal: create a semantically-equivalent block that uses some fixed number of physical registers ***k***
- If ***k*** < ***v***, there is another complication: spilling a register to memory may require **more registers**!
  - We have to account for this when we choose ***k***

Local Reg. Alloc.: Frequency Count Heuristic

Local Reg. Alloc.: Frequency Count Heuristic

- Key principle: the **most heavily-used** values should reside in registers

Local Reg. Alloc.: Frequency Count Heuristic

- Key principle: the **most heavily-used** values should reside in registers
  - hence the “frequency count” name: we just **count** how many times each virtual register is used
    - guaranteed to be finite + easy (it’s a basic block)

Local Reg. Alloc.: Frequency Count Heuristic

- Key principle: the **most heavily-used** values should reside in registers
  - hence the “frequency count” name: we just **count** how many times each virtual register is used
    - guaranteed to be finite + easy (it’s a basic block)
  - then, assign them to physical registers in frequency order
    - most commonly-used value in first register, etc.

Local Reg. Alloc.: Frequency Count Heuristic

- Key principle: the **most heavily-used** values should reside in registers
  - hence the “frequency count” name: we just **count** how many times each virtual register is used
    - guaranteed to be finite + easy (it’s a basic block)
  - then, assign them to physical registers in frequency order
    - most commonly-used value in first register, etc.
- Advantage of this heuristic: **easy to understand + implement**

Local Reg. Alloc.: Frequency Count Heuristic

- Key principle: the **most heavily-used** values should reside in registers
  - hence the "frequency count" name: we just **count** how many times each virtual register is used
    - guaranteed to be finite + easy (it's a basic block)
  - then, assign them to physical registers in frequency order
    - most commonly-used value in first register, etc.
- Advantage of this heuristic: **easy to understand + implement**
- Big disadvantage: it dedicates a physical register to one virtual register for the **entire basic block**
  - even if the virtual register is only used in the first half!

Local Reg. Alloc.: Greedy Algorithm

Local Reg. Alloc.: Greedy Algorithm

- This algorithm is “bottom up” because it makes decisions on an instruction-by-instruction basis

Local Reg. Alloc.: Greedy Algorithm

- This algorithm is “bottom up” because it makes decisions on an instruction-by-instruction basis
  - always trying to **maximize** the number of registers in use

Local Reg. Alloc.: Greedy Algorithm

- This algorithm is “bottom up” because it makes decisions on an instruction-by-instruction basis
  - always trying to **maximize** the number of registers in use
- Basic algorithm:

Local Reg. Alloc.: Greedy Algorithm

- This algorithm is “bottom up” because it makes decisions on an instruction-by-instruction basis
  - always trying to **maximize** the number of registers in use
- Basic algorithm:
  - assume that the physical registers are initially empty
    - initialize a **free list** with all physical registers

Local Reg. Alloc.: Greedy Algorithm

- This algorithm is “bottom up” because it makes decisions on an instruction-by-instruction basis
  - always trying to **maximize** the number of registers in use
- Basic algorithm:
  - assume that the physical registers are initially empty
    - initialize a **free list** with all physical registers
  - from the start of the block, assign virtual registers to physical registers; **remove** used physical registers from the free list
    - after last use of a virtual reg., return its phys. reg. to free list

Local Reg. Alloc.: Greedy Algorithm

- This algorithm is “bottom up” because it makes decisions on an instruction-by-instruction basis
  - always trying to **maximize** the number of registers in use
- Basic algorithm:
  - assume that the physical registers are initially empty
    - initialize a **free list** with all physical registers
  - from the start of the block, assign virtual registers to physical registers; **remove** used physical registers from the free list
    - after last use of a virtual reg., return its phys. reg. to free list
  - if the free list is empty, **spill** a register according to some policy
    - usually, the register whose next use is farthest in the future

Local Reg. Alloc.: Greedy Algorithm

- This algorithm is "bottom up" because it makes decisions on an instruction-by-instruction basis
  - always trying to **maximize** the number of registers in use
- Basic algorithm:
  - assume that the phy
    - initialize a **free l**
  - from the start of the block, assign virtual registers to physical registers; **remove** used physical registers from the free list
    - after last use of a virtual reg., return its phys. reg. to free list
  - if the free list is empty, **spill** a register according to some policy
    - usually, the register **whose next use is farthest in the future**

"whose next use is farthest in the future" is easy to pre-compute in a single pass (because it's a basic block)

Local Reg. Alloc.: Clean and Dirty Values

- There is one tricky complication that we need to deal with: all values in registers **don't need to be stored** on a spill!

Local Reg. Alloc.: Clean and Dirty Values

- There is one tricky complication that we need to deal with: all values in registers **don't need to be stored** on a spill!
  - for example, there is no need to store constants or values that originate from memory

Local Reg. Alloc.: Clean and Dirty Values

- There is one tricky complication that we need to deal with: all values in registers **don't need to be stored** on a spill!
  - for example, there is no need to store constants or values that originate from memory
- We refer to such values as ***clean*** values
  - ***dirty*** values do need to be stored

Local Reg. Alloc.: Clean and Dirty Values

- There is one tricky complication that we need to deal with: all values in registers **don't need to be stored** on a spill!
  - for example, there is no need to store constants or values that originate from memory
- We refer to such values as ***clean*** values
  - ***dirty*** values do need to be stored
- It is not obvious whether it is preferable to spill clean or dirty values: the answer is **context-dependent**
  - this is part of why the register allocation problem is so hard!

Local Reg. Alloc.: Clean and Dirty Values

- There is one tricky complication that we need to deal with: all values in registers **don't need to be stored** on a spill!
  - for example, there is no need to store constants or values that originate from memory
- We refer to such values as ***clean*** values
  - ***dirty*** values do need to be stored
- It is not obvious whether it is preferable to spill clean or dirty values: the answer is **context-dependent**
  - this is part of why the register allocation problem is so hard!
- Let's see an example…

Local Reg. Alloc.: Clean/Dirty Values Example

- Consider a two-register machine with this state:

phys. reg.	value/VR	clean?
r1	x1	clean
r2	x2	dirty

Local Reg. Alloc.: Clean/Dirty Values Example

- Consider a two-register machine with this state:

phys. reg.	value/VR	clean?
r1	x1	clean
r2	x2	dirty

- Suppose that the virtual registers used in the rest of the block are: x3, then x1, then x2 ("x3 x1 x2" is called a ***reference string***)

Local Reg. Alloc.: Clean/Dirty Values Example

- Consider a two-register machine with this state:

phys. reg.	value/VR	clean?
r1	x1	clean
r2	x2	dirty

- Suppose that the virtual registers used in the rest of the block are: x3, then x1, then x2 ("x3 x1 **x2**" is called a ***reference string***)

x2 is furthest use, so naïvely we should spill it:

```
store x2
load x3
load x2
```

Local Reg. Alloc.: Clean/Dirty Values Example

- Consider a two-register machine with this state:

phys. reg.	value/VR	clean?
r1	x1	clean
r2	x2	dirty

- Suppose that the virtual registers used in the rest of the block are: x3, then x1, then x2 ("x3 **x1** **x2**" is called a ***reference string***)

x2 is furthest use, so naïvely we should spill it:

```
store x2
load x3
load x2
```

but, if we spill **x1** instead, we get:

```
load x3 # overwrites x1
load x1
```

Local Reg. Alloc.: Clean/Dirty Values Example

- Consider a two-register machine with this state:

phys. reg.	value/VR	clean?
r1	x1	clean
r2	x2	dirty

- Suppose that the virtual registers used in the rest of the block are: x3, then x1, then x2 ("x3 **x1** **x2**" is called a ***reference string***)

x2 is furthest use, so naïvely we should spill it:

```
store x2
```

This example suggests that it might be better to **preferentially spill clean** values. Unfortunately, it's not that simple…

we get:

```
load x3 # overwrites x1
load x1
```

Local Reg. Alloc.: Clean/Dirty Values Example

- Returning to the original state:

phys. reg.	value/VR	clean?
r1	x1	clean
r2	x2	dirty

Local Reg. Alloc.: Clean/Dirty Values Example

- Returning to the original state:

phys. reg.	value/VR	clean?
r1	x1	clean
r2	x2	dirty

- Suppose that the reference string is "x3 x1 x3 x1 x2" instead. **In-class exercise**: turn to a neighbor and determine whether it's better to spill x1 or x2.

Local Reg. Alloc.: Clean/Dirty Values Example

- Returning to the original state:

phys. reg.	value/VR	clean?
r1	x1	clean
r2	x2	dirty

Spill x1:

```
load x3
load x1
load x3
load x1
```

Spill x2:

```
store x2
load x3
load x2
```

- Suppose that the reference string is "x3 x1 x3 x1 x2" instead. **In-class exercise**: turn to a neighbor and determine whether it's better to spill x1 or x2.

Local Reg. Alloc.: Clean/Dirty Values Example

- Returning to the original state:

phys. reg.	value/VR	clean?
r1	x1	clean
r2	x2	dirty

Spill x1:

```
load x3
load x1
load x3
load x1
```

Spill x2:

```
store x2
load x3
load x2
```

spilling the dirty value results in less spill code!

- Suppose that the reference string is "x3 x1 x3 x1 x2" instead. **In-class exercise**: turn to a neighbor and determine whether it's better to spill x1 or x2.

Local Reg. Alloc.: An Optimization

Local Reg. Alloc.: An Optimization

- We initially assumed that we **must store** every virtual register to memory at the end of a basic block
  - is this really true?

Local Reg. Alloc.: An Optimization

- We initially assumed that we **must store** every virtual register to memory at the end of a basic block
  - is this really true?
- No. In practice, many virtual registers contain values that are **dead** after the block (e.g., results of intermediate computations)

Local Reg. Alloc.: An Optimization

- We initially assumed that we **must store** every virtual register to memory at the end of a basic block
  - is this really true?
- No. In practice, many virtual registers contain values that are **dead** after the block (e.g., results of intermediate computations)
- As an optimization, we can **not store to memory** any value that is dead after the block.

Local Reg. Alloc.: An Optimization

- We initially assumed that we **must store** every virtual register to memory at the end of a basic block
  - is this really true?
- No. In practice, many virtual registers contain values that are **dead** after the block (e.g., results of intermediate computations)
- As an optimization, we can **not store to memory** any value that is dead after the block.
  - A **global liveness analysis** computes exactly this information
    - i.e., we can easily inspect its “live out” sets to determine which virtual registers (= temporaries/IR names) we don’t need to store

Local Reg. Alloc.: Trade-offs and Summary

Local Reg. Alloc.: Trade-offs and Summary

- In practice, the greedy algorithm produces **excellent** local allocations

Local Reg. Alloc.: Trade-offs and Summary

- In practice, the greedy algorithm produces **excellent** local allocations
  - though it is **NP-hard** to produce **optimal** local allocations for single basic blocks, because of clean vs dirty values

Local Reg. Alloc.: Trade-offs and Summary

- In practice, the greedy algorithm produces **excellent** local allocations
  - though it is **NP-hard** to produce **optimal** local allocations for single basic blocks, because of clean vs dirty values
  - frequency count allocator is **simpler**, but its allocations are usually worse

Local Reg. Alloc.: Trade-offs and Summary

- In practice, the greedy algorithm produces **excellent** local allocations
  - though it is **NP-hard** to produce **optimal** local allocations for single basic blocks, because of clean vs dirty values
  - frequency count allocator is **simpler**, but its allocations are usually worse
- If you are going to implement a local allocator in PA4, this greedy strategy is a good option
  - the textbook has a more-detailed version of this algorithm
  - if you want to do register allocation, I recommend building a local allocator first, regardless

Regional Register Allocation

Regional Register Allocation

- Just as in IR-level optimization, we might think that we can extend our **local** register allocation algorithm to work at the **regional** level
  - with a few tweaks

Regional Register Allocation

- Just as in IR-level optimization, we might think that we can extend our **local** register allocation algorithm to work at the **regional** level
  - with a few tweaks
- Most obvious improvement: keep **live variables** in registers between blocks (instead of storing them to memory)

Regional Register Allocation

- Just as in IR-level optimization, we might think that we can extend our **local** register allocation algorithm to work at the **regional** level
  - with a few tweaks
- Most obvious improvement: keep **live variables** in registers between blocks (instead of storing them to memory)
  - If we've implemented the optimization I described a few slides ago to avoid storing *dead* variables, then this should be easy
    - we are already running the liveness analysis!

Regional Register Allocation

- Just as in IR-level optimization, we might think that we can extend our **local** register allocation algorithm to work at the **regional** level
  - with a few tweaks
- Most obvious improvement: keep **live variables** in registers between blocks (instead of storing them to memory)
  - If we've implemented the optimization I described a few slides ago to avoid storing *dead* variables, then this should be easy
    - we are already running the liveness analysis!
- Unfortunately, this is a bit of a **false hope**.

Regional Register Allocation

- Just as in IR-level optimization, we might think that we can extend our **local** register allocation algorithm to work at the **regional** level
  - with a few tweaks
- Most obvious improvement: keep **live variables** in registers between blocks (instead of storing them to memory)
  - If we've implemented the optimization I described a few slides ago to avoid storing *dead* variables, then this should be easy
    - we are already running the liveness analysis!
- Unfortunately, this is a bit of a **false hope**.
  - There are a lot of complications that arise at basic block boundaries! Let's take a look at some examples to see why.

Regional Register Allocation: Strawman

- Consider this algorithm for a simple regional register allocator:

Regional Register Allocation: Strawman

- Consider this algorithm for a simple regional register allocator:
- For each basic block **b**, in CFG order:

Regional Register Allocation: Strawman

- Consider this algorithm for a simple regional register allocator:
- For each basic block **b**, in CFG order:
  - Run the greedy register allocation algorithm we just saw on **b**

Regional Register Allocation: Strawman

- Consider this algorithm for a simple regional register allocator:
- For each basic block **b**, in CFG order:
  - Run the greedy register allocation algorithm we just saw on **b**
  - Run a liveness analysis and compute a set (of virtual registers!) that are live when **b** exits

Regional Register Allocation: Strawman

- Consider this algorithm for a simple regional register allocator:
- For each basic block **b**, in CFG order:
  - Run the greedy register allocation algorithm we just saw on **b**
  - Run a liveness analysis and compute a set (of virtual registers!) that are live when **b** exits
    - note that this assumes we have a **global** liveness analysis!

Regional Register Allocation: Strawman

- Consider this algorithm for a simple regional register allocator:
- For each basic block **b**, in CFG order:
  - Run the greedy register allocation algorithm we just saw on **b**
  - Run a liveness analysis and compute a set (of virtual registers!) that are live when **b** exits
    - note that this assumes we have a **global** liveness analysis!
  - For each successor of **b**, start the greedy register allocation algorithm in a state where the those virtual registers are already allocated to the same physical registers that we allocated them to in **b**

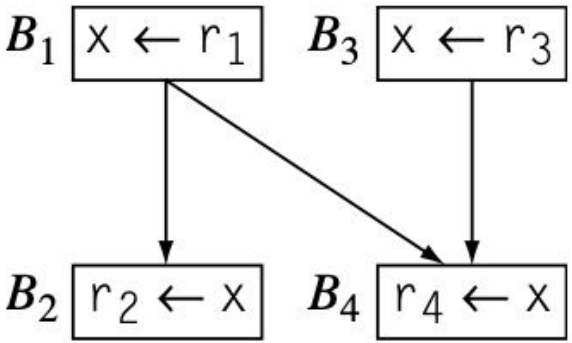
Regional Register Allocation: Strawman

- Consider this algorithm for a simple regional register allocator:
- For each basic block **b**, in CFG order:
  - Run the greedy register allocation algorithm we just saw on **b**
  - Run a liveness analysis and compute a set (of virtual registers!) that are live when **b** exits
    - note that this assumes we have a **global** liveness analysis!
  - For each successor of **b**, start the greedy register allocation algorithm in a state where the those virtual registers are already allocated to the same physical registers that we allocated them to in **b**

Why won't this work?

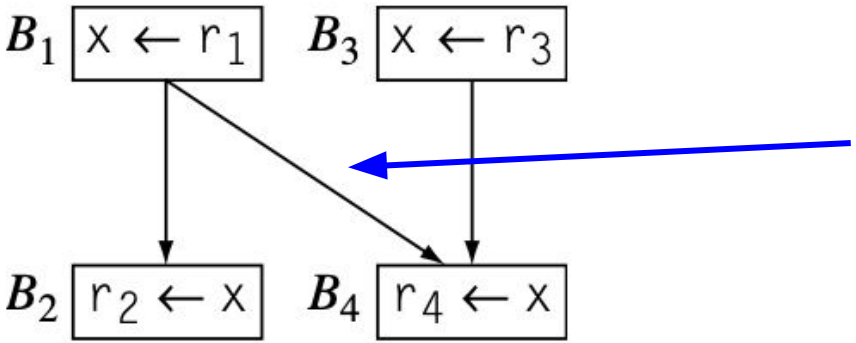
Regional Register Allocation: Strawman

- Consider this CFG (with some specific register allocations returned by the local allocator and values in the basic blocks):



Regional Register Allocation: Strawman

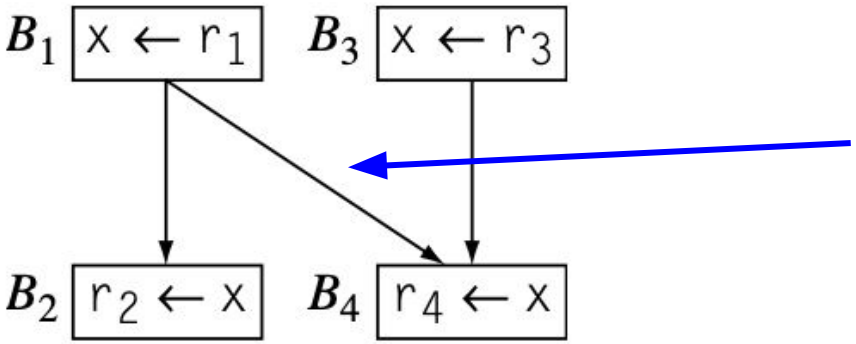
- Consider this CFG (with some specific register allocations returned by the local allocator and values in the basic blocks):



this edge B_1 -> B_4 is a ***critical*** edge: there is no safe choice for the register to store x, because of B_2 and B_3

Regional Register Allocation: Strawman

- Consider this CFG (with some specific register allocations returned by the local allocator and values in the basic blocks):



this edge B_1 -> B_4 is a ***critical*** edge: there is no safe choice for the register to store x, because of B_2 and B_3

- e.g., consider adding copy operations to keep x in a register. Where would they go?

Regional Register Allocation: Strawman

- Consider this CFG (with some specific register allocations returned by the local allocator and values in the basic blocks):

B_1 $\boxed{x \leftarrow r_1}$ B_3 $\boxed{x \leftarrow r_3}$

this edge B_1 -> B_4 is a ***critical***

afe choice

store x,

B_3

dding copy

eep x in a

e would

they go?

Critical edges are just one reason that extending a local allocator beyond a single block is a bad idea. Instead, a global allocator should **coordinate** allocation decisions.

- it turns out that global allocators are fundamentally quite different than local allocators

Trivia Break: Popular Culture

This American science fiction media franchise began with a television show in the mid-1960s. The franchise is noted for its cultural influence beyond works of science fiction and for its progressive stances on civil rights; for example, in the 1960s, it was one of the first shows with a multiracial cast on American television. The franchise contains 11 spin-off television series and a film franchise; further adaptations also exist in several media. The series greatly influenced public interest in the U.S. Space Program and in education on the topic of space exploration. For example, NASA named one of its space shuttles (now on display at the Intrepid Museum in NYC) after something from this series.

Trivia Break: Physics

The creator of this speculative theory stated in an email to William Shatner (who played Captain Kirk in the *Star Trek* original series) that his theory was directly inspired by the term used in the show. The theory posits a speculative warp drive idea according to which a spacecraft could achieve apparent faster-than-light travel by contracting space in front of it and expanding space behind it, under the assumption that a configurable energy-density field lower than that of vacuum (that is, negative mass) could be created. The idea remains a hypothetical concept with seemingly difficult problems, although the amount of energy required is no longer thought to be unobtainably large.

Global Register Allocation

Global Register Allocation

- We're going to cover one global register allocation algorithm: **reduction to graph coloring**

Global Register Allocation

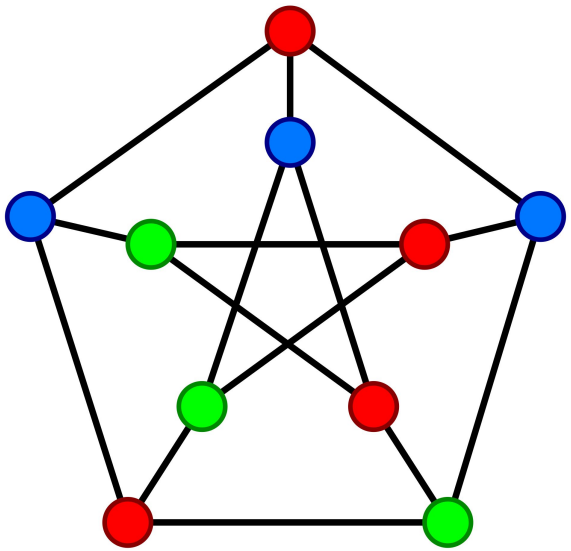
- We're going to cover one global register allocation algorithm: **reduction to graph coloring**
  - graph coloring is a famous NP-complete problem

Global Register Allocation

- We're going to cover one global register allocation algorithm: **reduction to graph coloring**
  - graph coloring is a famous NP-complete problem
  - specifically, we're interested in the ***vertex coloring*** problem: given a graph and a set of labels ("colors"), find an assignment of labels to nodes (a "***coloring***") such that no two adjacent nodes have the same color

Global Register Allocation

- We're going to cover one global register allocation algorithm: **reduction to graph coloring**
  - graph coloring is a famous NP-complete problem
  - specifically, we're interested in the ***vertex coloring*** problem: given a graph and a set of labels ("colors"), find an assignment of labels to nodes (a "***coloring***") such that no two adjacent nodes have the same color



Global Reg. Alloc.: Reduction Setup

- To do explain this reduction, we need to cover two things:

Global Reg. Alloc.: Reduction Setup

- To do explain this reduction, we need to cover two things:
  - How do we **map** the global register allocation problem to a graph coloring problem?

Global Reg. Alloc.: Reduction Setup

- To do explain this reduction, we need to cover two things:
  - How do we **map** the global register allocation problem to a graph coloring problem?
    - We will extend our notion of liveness to ***live ranges*** to facilitate this transformation

Global Reg. Alloc.: Reduction Setup

- To do explain this reduction, we need to cover two things:
  - How do we **map** the global register allocation problem to a graph coloring problem?
    - We will extend our notion of liveness to ***live ranges*** to facilitate this transformation
    - We will use those live ranges to construct an ***interference graph***

Global Reg. Alloc.: Reduction Setup

- To do explain this reduction, we need to cover two things:
  - How do we **map** the global register allocation problem to a graph coloring problem?
    - We will extend our notion of liveness to ***live ranges*** to facilitate this transformation
    - We will use those live ranges to construct an ***interference graph***
  - Once we have a graph coloring problem, how do we actually **solve** it? It's NP-complete, after all…

Global Reg. Alloc.: Reduction Setup

- To do explain this reduction, we need to cover two things:
  - How do we **map** the global register allocation problem to a graph coloring problem?
    - We will extend our notion of liveness to ***live ranges*** to facilitate this transformation
    - We will use those live ranges to construct an ***interference graph***
  - Once we have a graph coloring problem, how do we actually **solve** it? It's NP-complete, after all…
    - We will see an algorithm that takes advantage of a special property of our graph: we can spill registers to **simplify** it

Global Reg. Alloc.: Live Ranges

Global Reg. Alloc.: Live Ranges

- A ***live range*** consists of a set of definitions and uses that are related to each other because their values flow together:

Global Reg. Alloc.: Live Ranges

- A ***live range*** consists of a set of definitions and uses that are related to each other because their values flow together:
  - for each use, every definition that can reach that use is in the same live range as the use

Global Reg. Alloc.: Live Ranges

- A ***live range*** consists of a set of definitions and uses that are related to each other because their values flow together:
  - for each use, every definition that can reach that use is in the same live range as the use
  - for each definition, every use that can refer to the result of the definition is in the same live range as the definition

Global Reg. Alloc.: Live Ranges

- A ***live range*** consists of a set of definitions and uses that are related to each other because their values flow together:
  - for each use, every definition that can reach that use is in the same live range as the use
  - for each definition, every use that can refer to the result of the definition is in the same live range as the definition
- The set of live ranges is **distinct** from the set of variables and the set of values

Global Reg. Alloc.: Live Ranges

- A ***live range*** consists of a set of definitions and uses that are related to each other because their values flow together:
  - for each use, every definition that can reach that use is in the same live range as the use
  - for each definition, every use that can refer to the result of the definition is in the same live range as the definition
- The set of live ranges is **distinct** from the set of variables and the set of values
  - Every value computed in the code is part of some live range, even if it has no name in the original source code

Global Reg. Alloc.: Live Ranges

- A ***live range*** consists of a set of definitions and uses that are related to each other because their values flow together:
  - for each use, every definition that can reach that use is in the same live range as the use
  - for each definition, every use that can refer to the result of the definition is in the same live range as the definition
- The set of live ranges is **distinct** from the set of variables and the set of values
  - Every value computed in the code is part of some live range, even if it has no name in the original source code
- To make this concrete, let's start with **local live ranges**…

Global Reg. Alloc.: Local Live Ranges

- In straightline code, we can represent a live range as an **interval** [i,j] where operation i defines it and operation j is its last use

Global Reg. Alloc.: Local Live Ranges

- In straightline code, we can represent a live range as an **interval** [i,j] where operation i defines it and operation j is its last use
  - We'll need a more complex notation later…

Global Reg. Alloc.: Local Live Ranges

- In straightline code, we can represent a live range as an **interval** [i,j] where operation i defines it and operation j is its last use
  - We'll need a more complex notation later…
- For now, consider this example program:

```
1  loadI   ...        ⇒ r_arp
2  loadAI  r_arp,@a   ⇒ r_a
3  loadI   2          ⇒ r_2
4  loadAI  r_arp,@b   ⇒ r_b
5  loadAI  r_arp,@c   ⇒ r_c
6  loadAI  r_arp,@d   ⇒ r_x
7  mult    r_a,r_2    ⇒ r_a
8  mult    r_a,r_b    ⇒ r_a
9  mult    r_a,r_c    ⇒ r_a
10 mult    r_a,r_d    ⇒ r_a
11 storeAI r_a        ⇒ r_arp,@a
```

Global Reg. Alloc.: Loca

What are the live ranges for some of these (virtual) registers?

- In straightline code, we can represent a live range as an **interval** [i,j] where operation i defines it and operation j is its last use
  - We'll need a more complex notation later…
- For now, consider this example program:

```
1  loadI   ...        ⇒ r_arp
2  loadAI  r_arp,@a   ⇒ r_a
3  loadI   2          ⇒ r_2
4  loadAI  r_arp,@b   ⇒ r_b
5  loadAI  r_arp,@c   ⇒ r_c
6  loadAI  r_arp,@d   ⇒ r_x
7  mult    r_a,r_2    ⇒ r_a
8  mult    r_a,r_b    ⇒ r_a
9  mult    r_a,r_c    ⇒ r_a
10 mult    r_a,r_d    ⇒ r_a
11 storeAI r_a        ⇒ r_arp,@a
```

Global Reg. Alloc.: Local Live Ranges

- In straightline code, we can represent a live range as an **interval** [i,j] where operation i defines it and operation j is its last use
  - We'll need a more complex notation later…
- For now, consider this example program:

```
1  loadI   ...        ⇒ r_arp
2  loadAI  r_arp,@a   ⇒ r_a
3  loadI   2          ⇒ r_2
4  loadAI  r_arp,@b   ⇒ r_b
5  loadAI  r_arp,@c   ⇒ r_c
6  loadAI  r_arp,@d   ⇒ r_x
7  mult    r_a,r_2    ⇒ r_a
8  mult    r_a,r_b    ⇒ r_a
9  mult    r_a,r_c    ⇒ r_a
10 mult    r_a,r_d    ⇒ r_a
11 storeAI r_a        ⇒ r_arp,@a
```

	Register	Interval
1	r_{arp}	[1,11]
2	r_a	[2,7]
3	r_a	[7,8]
4	r_a	[8,9]
5	r_a	[9,10]
6	r_a	[10,11]
7	r_2	[3,7]
8	r_b	[4,8]
9	r_c	[5,9]
10	r_d	[6,10]

Global Reg. Alloc.: Global Live Ranges

Global Reg. Alloc.: Global Live Ranges

- The interval notation is fine for local live ranges, but will **not suffice** for global live ranges

Global Reg. Alloc.: Global Live Ranges

- The interval notation is fine for local live ranges, but will **not suffice** for global live ranges
  - A virtual register may have uses in more than one block

Global Reg. Alloc.: Global Live Ranges

- The interval notation is fine for local live ranges, but will **not suffice** for global live ranges
  - A virtual register may have uses in more than one block
  - More than one definition may flow to each use (via phi functions)

Global Reg. Alloc.: Global Live Ranges

- The interval notation is fine for local live ranges, but will **not suffice** for global live ranges
  - A virtual register may have uses in more than one block
  - More than one definition may flow to each use (via phi functions)
- A global live range is a ***web*** of definitions and uses:

Global Reg. Alloc.: Global Live Ranges

- The interval notation is fine for local live ranges, but will **not suffice** for global live ranges
  - A virtual register may have uses in more than one block
  - More than one definition may flow to each use (via phi functions)
- A global live range is a ***web*** of definitions and uses:
  - For a use *u* in live range LR_i, LR_i must include every definition *d* that reaches *u*

Global Reg. Alloc.: Global Live Ranges

- The interval notation is fine for local live ranges, but will **not suffice** for global live ranges
  - A virtual register may have uses in more than one block
  - More than one definition may flow to each use (via phi functions)
- A global live range is a ***web*** of definitions and uses:
  - For a use *u* in live range LR_i, LR_i must include every definition *d* that reaches *u*
  - For each definition *d* in LR_i, LR_i must include every use *u* that *d* reaches

Global Reg. Alloc.: Global Live Ranges

- The interval notation is fine for local live ranges, but will **not suffice** for global live ranges
  - A virtual register may have uses in more than one block
  - More than one definition may flow to each use (via phi functions)
- A global live range is a ***web*** of definitions and uses:
  - For a use *u* in live range LR_i, LR_i must include every definition *d* that reaches *u*
  - For each definition *d* in LR_i, LR_i must include every use *u* that *d* reaches
  - Logically, a global live range is a **closure** of definitions and uses

Global Reg. Alloc.: Live Ranges + SSA

- Note how in the previous example, there were several different live ranges for r_a

Global Reg. Alloc.: Live Ranges + SSA

- Note how in the previous example, there were several different live ranges for r_a
  - this virtual register was **re-defined** several times

Global Reg. Alloc.: Live Ranges + SSA

- Note how in the previous example, there were several different live ranges for r_a
  - this virtual register was **re-defined** several times
  - **insight**: we can simplify live range construction by converting the input to **SSA form** to guarantee one live range per variable

Global Reg. Alloc.: Live Ranges + SSA

- Note how in the previous example, there were several different live ranges for r_a
  - this virtual register was **re-defined** several times
  - **insight**: we can simplify live range construction by converting the input to **SSA form** to guarantee one live range per variable
    - I will assume this from now on

Global Reg. Alloc.: Building Global Live Ranges

- To **build** global live ranges, start with the target program in SSA form

Global Reg. Alloc.: Building Global Live Ranges

- To **build** global live ranges, start with the target program in SSA form
- Then, use a **disjoint-set union-find** algorithm to combine SSA names into live ranges

Global Reg. Alloc.: Building Global Live Ranges

- To **build** global live ranges, start with the target program in SSA form
- Then, use a **disjoint-set union-find** algorithm to combine SSA names into live ranges
  - this algorithm assumes access to a ***disjoint-set forest***, a data structure that is specialized to this kind of problem

Global Reg. Alloc.: Building Global Live Ranges

- To **build** global live ranges, start with the target program in SSA form
- Then, use a **disjoint-set union-find** algorithm to combine SSA names into live ranges
  - this algorithm assumes access to a ***disjoint-set forest***, a data structure that is specialized to this kind of problem
    - O(1) inserts, O(α(n)) amortized search

Global Reg. Alloc.: Building Global Live Ranges

- To **build** global live ranges, start with the target program in SSA form
- Then, use a **disjoint-set union-find** algorithm to combine SSA names into live ranges
  - this algorithm assumes access to a ***disjoint-set forest***, a data structure that is specialized to this kind of problem
    - O(1) inserts, O(α(n)) amortized search
      - α here is the ***inverse Ackermann function***, which is an extraordinarily slow-growing function

Global Reg. Alloc.: Building Global Live Ranges

- To **build** global live ranges, start with the target program in SSA form
- Then, use a **disjoint-set union-find** algorithm to combine SSA names into live ranges
  - this algorithm assumes access to a ***disjoint-set forest***, a data structure that is specialized to this kind of problem
    - O(1) inserts, O(α(n)) amortized search
      - α here is the ***inverse Ackermann function***, which is an extraordinarily slow-growing function
  - this data structure is a key component in other algorithms, such as Kruskal's (graph minimum spanning trees) and unification (for equation solving)

Global Reg. Alloc.: Building Global Live Ranges

- Algorithm:

Global Reg. Alloc.: Building Global Live Ranges

- Algorithm:
  - start with a set of single-element sets
    - one for each SSA name

Global Reg. Alloc.: Building Global Live Ranges

- Algorithm:
  - start with a set of single-element sets
    - one for each SSA name
  - at each phi function invocation, union together the three sets
    - one set for each parameter and one set for the result

Global Reg. Alloc.: Building Global Live Ranges

- Algorithm:
  - start with a set of single-element sets
    - one for each SSA name
  - at each phi function invocation, union together the three sets
    - one set for each parameter and one set for the result
    - warning: this operation is fast on a disjoint-set forest, but if you implement this another way it could be very slow

Global Reg. Alloc.: Building Global Live Ranges

- Algorithm:
  - start with a set of single-element sets
    - one for each SSA name
  - at each phi function invocation, union together the three sets
    - one set for each parameter and one set for the result
    - warning: this operation is fast on a disjoint-set forest, but if you implement this another way it could be very slow
  - after all phi functions have been processed, the resulting sets exactly correspond to live ranges

Global Reg. Alloc.: Building a Graph to Color

- Now that we have live ranges, we need to construct a graph that will help us with register allocation: the interference graph.

Global Reg. Alloc.: Building a Graph to Color

- Now that we have live ranges, we need to construct a graph that will help us with register allocation: the interference graph.
- The ***interference graph*** is an undirected graph used to represent interference among the values of a program

Global Reg. Alloc.: Building a Graph to Color

- Now that we have live ranges, we need to construct a graph that will help us with register allocation: the interference graph.
- The ***interference graph*** is an undirected graph used to represent interference among the values of a program
  - The nodes represent program values

Global Reg. Alloc.: Building a Graph to Color

- Now that we have live ranges, we need to construct a graph that will help us with register allocation: the interference graph.
- The ***interference graph*** is an undirected graph used to represent interference among the values of a program
  - The nodes represent program values
  - The edges represent interference among values

Global Reg. Alloc.: Building a Graph to Color

- Now that we have live ranges, we need to construct a graph that will help us with register allocation: the interference graph.
- The ***interference graph*** is an undirected graph used to represent interference among the values of a program
  - The nodes represent program values
  - The edges represent interference among values
    - That is, there is an edge from node n1 to node n2 if the values corresponding to n1 and n2 are **simultaneously live**

Global Reg. Alloc.: Building a Graph to Color

- Now that we have live ranges, we need to construct a graph that will help us with register allocation: the interference graph.
- The ***interference graph*** is an undirected graph used to represent interference among the values of a program
  - The nodes represent program values
  - The edges represent interference among values
    - That is, there is an edge from node n1 to node n2 if the values corresponding to n1 and n2 are **simultaneously live**
      - We can easily derive this information from the live ranges of the values for n1 and n2

Global Reg. Alloc.: Interference Graph Example

x <- 1
y <- 2
z <- x + y
t <- y
u <- x + t
print z
print t
print u

Global Reg. Alloc.: Interference Graph Example

Live ranges:

	x	y	z	t	u
x <- 1					
y <- 2					
z <- x + y					
t <- y					
u <- x + t					
print z					
print t					
print u					

Global Reg. Alloc.: Interference Graph Example

Live ranges:

	x	y	z	t	u
x <- 1					
y <- 2					
z <- x + y					
t <- y					
u <- x + t					
print z					
print t					
print u					

Global Reg. Alloc.: Interference Graph Example

Live ranges:

	x	y	z	t	u
x <- 1					
y <- 2					
z <- x + y					
t <- y					
u <- x + t					
print z					
print t					
print u					

Global Reg. Alloc.: Interference Graph Example

Live ranges:

	x	y	z	t	u
x <- 1					
y <- 2					
z <- x + y					
t <- y					
u <- x + t					
print z					
print t					
print u					

Global Reg. Alloc.: Interference Graph Example

Live ranges:

	x	y	z	t	u
x <- 1					
y <- 2					
z <- x + y					
t <- y					
u <- x + t					
print z					
print t					
print u					

Global Reg. Alloc.: Interference Graph Example

Live ranges:

	x	y	z	t	u
x <- 1					
y <- 2					
z <- x + y					
t <- y					
u <- x + t					
print z					
print t					
print u					

Global Reg. Alloc.: Interference Graph Example

Live ranges:

	x	y	z	t	u
x <- 1					
y <- 2					
z <- x + y					
t <- y					
u <- x + t					
print z					
print t					
print u					

Global Reg. Alloc.: Interference Graph Example

Live ranges:

	x	y	z	t	u
x <- 1					
y <- 2					
z <- x + y					
t <- y					
u <- x + t					
print z					
print t					
print u					

Global Reg. Alloc.: Interference Graph Example

Live ranges:

	x	y	z	t	u
x <- 1					
y <- 2					
z <- x + y					
t <- y					
u <- x + t					
print z					
print t					
print u					

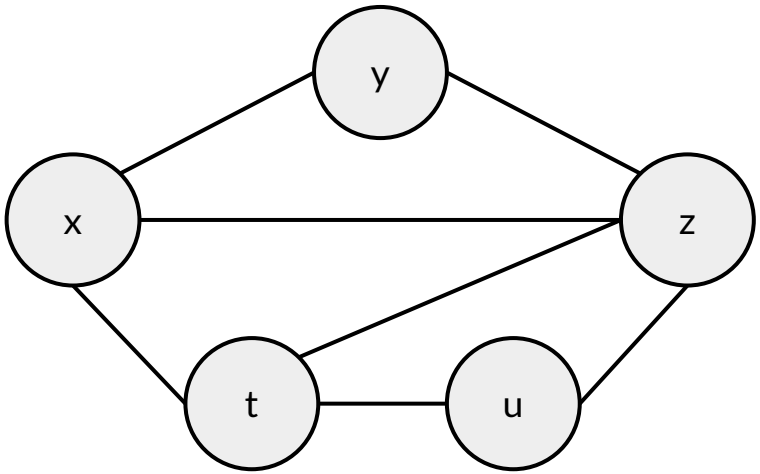
What's the interference graph? How would you derive it from this table?

Global Reg. Alloc.: Interference Graph Example

Live ranges:

	x	y	z	t	u
x <- 1					
y <- 2					
z <- x + y					
t <- y					
u <- x + t					
print z					
print t					
print u					

What's the interference graph? How would you derive it from this table?



Global Reg. Alloc.: Coloring the Graph

Global Reg. Alloc.: Coloring the Graph

- Graph coloring can be used to allocate registers to values by trying to color the interference graph with **as many colors as there are registers** in the target machine

Global Reg. Alloc.: Coloring the Graph

- Graph coloring can be used to allocate registers to values by trying to color the interference graph with **as many colors as there are registers** in the target machine
  - This is **not always possible**, in which case some values must be spilled (i.e. stored in memory)

Global Reg. Alloc.: Coloring the Graph

- Graph coloring can be used to allocate registers to values by trying to color the interference graph with **as many colors as there are registers** in the target machine
  - This is **not always possible**, in which case some values must be spilled (i.e. stored in memory)
  - We can map “spilling a value” back into the graph to **simplify** the interference graph (and therefore the graph coloring problem)

Global Reg. Alloc.: Coloring the Graph

- Graph coloring can be used to allocate registers to values by trying to color the interference graph with **as many colors as there are registers** in the target machine
  - This is **not always possible**, in which case some values must be spilled (i.e. stored in memory)
  - We can map "spilling a value" back into the graph to **simplify** the interference graph (and therefore the graph coloring problem)
    - To be clear: this approach is **heuristic** (graph coloring is NP-complete). You could solve the graph coloring problem here using any graph coloring algorithm.

Global Reg. Alloc.: Coloring by Simplification

- Assume we are coloring a graph G with K colors

Global Reg. Alloc.: Coloring by Simplification

- Assume we are coloring a graph G with K colors
- Coloring by simplification works as follows:

Global Reg. Alloc.: Coloring by Simplification

- Assume we are coloring a graph G with K colors
- Coloring by simplification works as follows:
  - as long as the graph G has at least one node n with **less than K neighbors**, n is removed from G, and coloring proceeds with that simplified graph

Global Reg. Alloc.: Coloring by Simplification

- Assume we are coloring a graph G with K colors
- Coloring by simplification works as follows:
  - as long as the graph G has at least one node n with **less than K neighbors**, n is removed from G, and coloring proceeds with that simplified graph
    - we assign n a *specific* color later; we remove it here because we know that coloring it is *possible*

Global Reg. Alloc.: Coloring by Simplification

- Assume we are coloring a graph G with K colors
- Coloring by simplification works as follows:
  - as long as the graph G has at least one node n with **less than K neighbors**, n is removed from G, and coloring proceeds with that simplified graph
    - we assign n a *specific* color later; we remove it here because we know that coloring it is *possible*
- More formally, if the simplified graph is ***K-colorable***, then so is G: since n has less than K neighbors, those use at most K-1 colors, and there is therefore at least one color available for n.

Global Reg. Alloc.: Coloring by Simplification

- During simplification, it is possible to reach a point where **all nodes have at least K neighbors**

Global Reg. Alloc.: Coloring by Simplification

- During simplification, it is possible to reach a point where **all nodes have at least K neighbors**
  - When this occurs, a node must be chosen and its value must be **spilled**

Global Reg. Alloc.: Coloring by Simplification

- During simplification, it is possible to reach a point where **all nodes have at least K neighbors**
  - When this occurs, a node must be chosen and its value must be **spilled**
  - Then, we can remove its node from the graph

Global Reg. Alloc.: Coloring by Simplification

- During simplification, it is possible to reach a point where **all nodes have at least K neighbors**
  - When this occurs, a node must be chosen and its value must be **spilled**
  - Then, we can remove its node from the graph
- When colors are assigned to nodes after simplification is complete, it can happen that a node initially designated as spilled can be colored because its neighbors do not use all available colors

Global Reg. Alloc.: Coloring by Simplification

- During simplification, it is possible to reach a point where **all nodes have at least K neighbors**
  - When this occurs, a node must be chosen and its value must be **spilled**
  - Then, we can remove its node from the graph
- When colors are assigned to nodes after simplification is complete, it can happen that a node initially designated as spilled can be colored because its neighbors do not use all available colors
  - When this happens, the potential spill is not turned into an actual spill

Global Reg. Alloc.: Coloring by Simplification

- During simplification, it is possible to reach a point where **all nodes have at least K neighbors**
  - When this occurs, a node must be chosen and its value must be **spilled**
  - Then, we can remove its node from the graph
- When colors are assigned to nodes after simplification is complete, it can happen that a node initially designated as spilled can be colored because its neighbors do not use all available colors
  - When this happens, the potential spill is not turned into an actual spill
  - This technique is known as ***optimistic coloring***

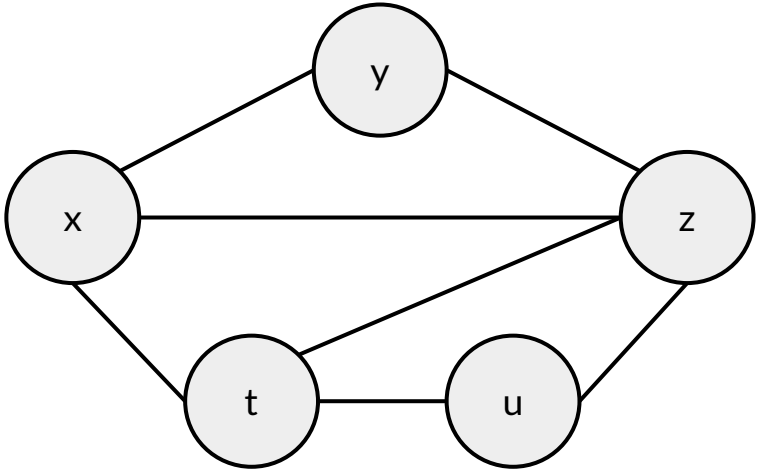
Global Reg. Alloc.: Coloring by Simplification

- During simplification, it is possible to reach a point where **all nodes have at lea**
  - When t e **spilled**
  - Then, w
- When colo e, it can happen that a node initially designated as spilled can be colored because its neighbors do not use all available colors
  - When this happens, the potential spill is not turned into an actual spill
  - This technique is known as ***optimistic coloring***

When a node is really spilled, the program has to be rewritten, which changes the interference graph. When this happens, the allocation process must be **restarted completely**. In practice, it converges in one or two iterations in most cases.

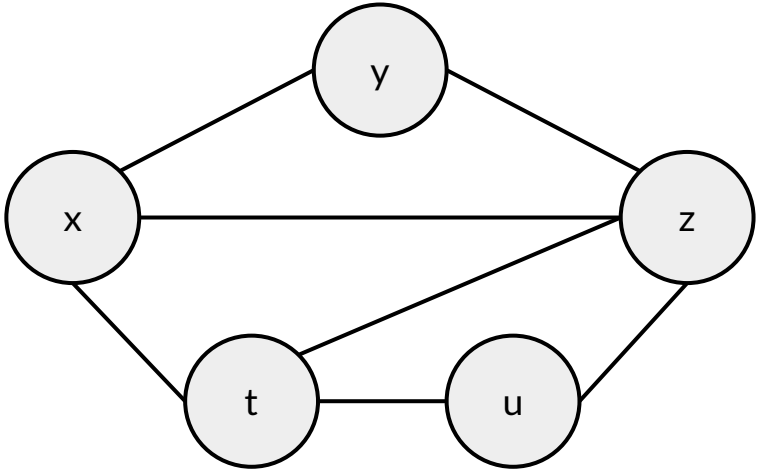
Coloring by Simplification Example

Consider the graph from earlier:



Coloring by Simplification Example

Consider the graph from earlier:

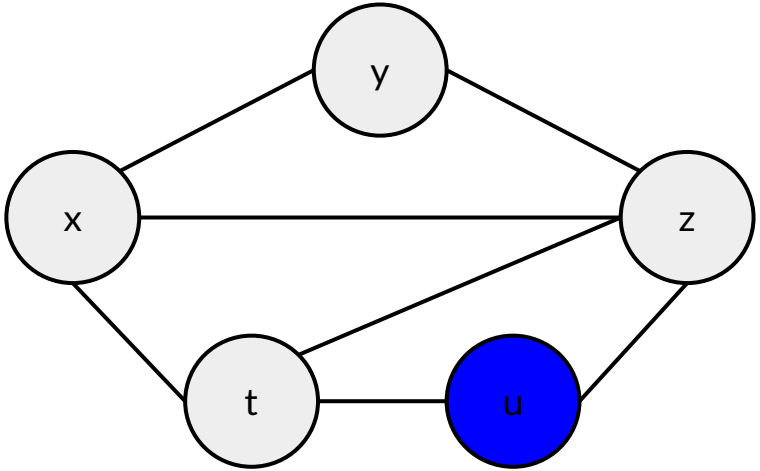


Let's try to fit these values into 3 registers:

- **r1**
- **r2**
- **r3**

Coloring by Simplification Example

Consider the graph from earlier:

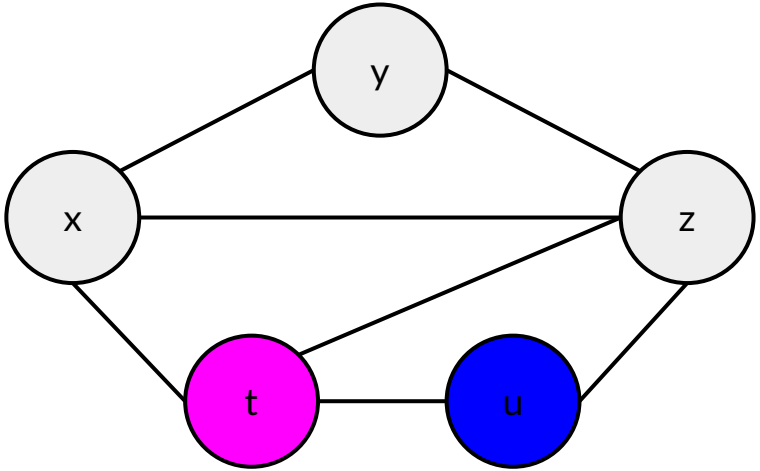


Let's try to fit these values into 3 registers:

- **r1**
- **r2**
- **r3**

Coloring by Simplification Example

Consider the graph from earlier:

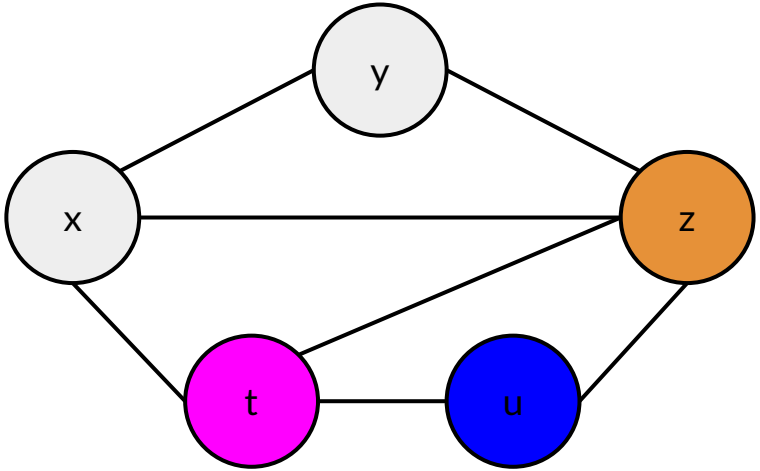


Let's try to fit these values into 3 registers:

- **r1**
- **r2**
- **r3**

Coloring by Simplification Example

Consider the graph from earlier:

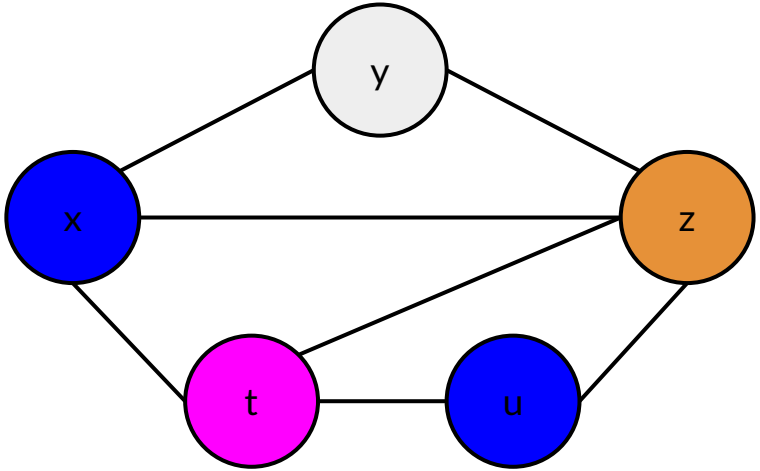


Let's try to fit these values into 3 registers:

- **r1**
- **r2**
- **r3**

Coloring by Simplification Example

Consider the graph from earlier:

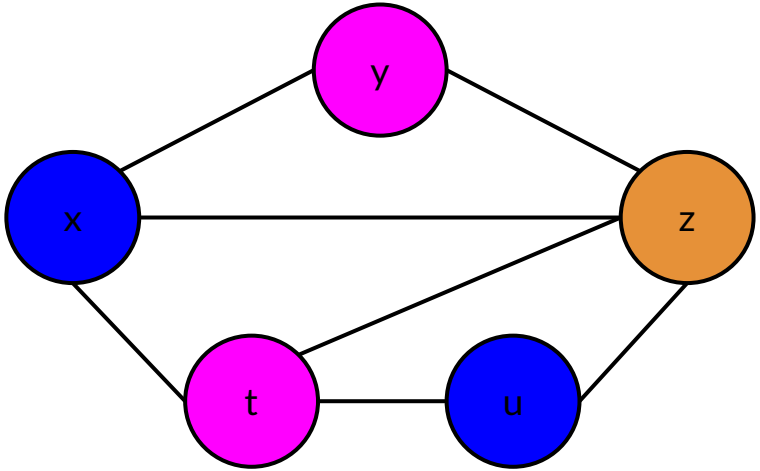


Let's try to fit these values into 3 registers:

- **r1**
- **r2**
- **r3**

Coloring by Simplification Example

Consider the graph from earlier:



Let's try to fit these values into 3 registers:

- **r1**
- **r2**
- **r3**

Global Reg. Alloc.: Coloring Complications

Global Reg. Alloc.: Coloring Complications

- It is sometimes useful to ***coalesce*** two SSA names that do not share an edge into a single name (and then put them in the same register)

Global Reg. Alloc.: Coloring Complications

- It is sometimes useful to ***coalesce*** two SSA names that do not share an edge into a single name (and then put them in the same register)
  - unfortunately, proving that this is safe is tough

Global Reg. Alloc.: Coloring Complications

- It is sometimes useful to ***coalesce*** two SSA names that do not share an edge into a single name (and then put them in the same register)
  - unfortunately, proving that this is safe is tough
- Similarly, there are some situations where ***splitting a live range*** can take a graph that is not K-colorable and make it K-colorable

Global Reg. Alloc.: Coloring Complications

- It is sometimes useful to ***coalesce*** two SSA names that do not share an edge into a single name (and then put them in the same register)
  - unfortunately, proving that this is safe is tough
- Similarly, there are some situations where ***splitting a live range*** can take a graph that is not K-colorable and make it K-colorable
  - "splitting a live range" = storing the value to memory and the retrieving it later, effectively creating two new live ranges

Global Reg. Alloc.: Coloring Complications

- It is sometimes useful to ***coalesce*** two SSA names that do not share an edge into a single name (and then put them in the same register)
  - unfortunately, proving that this is safe is tough
- Similarly, there are some situations where ***splitting a live range*** can take a graph that is not K-colorable and make it K-colorable
  - “splitting a live range” = storing the value to memory and the retrieving it later, effectively creating two new live ranges
  - again, no good, general-case heuristics :(

Global Reg. Alloc.: Coloring Complications

- It is sometimes useful to ***coalesce*** two SSA names that do not share an edge into a single name (and then put them in the same register)
  - unfortunately, proving that this is safe is tough
- Similarly, there are some situations where ***splitting a live range*** can take a graph that is not K-colorable and make it K-colorable
  - "splitting a live range" = storing the value to memory and the retrieving it later, effectively creating two new live ranges
  - again, no good, general-case heuristics :(
- It's common that you may need to put some values in specific registers, e.g. to adhere to calling conventions

Global Reg. Alloc.: Coloring Complications

- It is sometimes useful to ***coalesce*** two SSA names that do not share an edge into a single name (and then put them in the same register)
  - unfortunately, proving that this is safe is tough
- Similarly, there are some situations where ***splitting a live range*** can take a graph that is not K-colorable and make it K-colorable
  - "splitting a live range" = storing the value to memory and the retrieving it later, effectively creating two new live ranges
  - again, no good, general-case heuristics :(
- It's common that you may need to put some values in specific registers, e.g. to adhere to calling conventions
  - this is easy to handle: just **pre-color** those nodes in the graph

Global Register Allocation: Summary

Global Register Allocation: Summary

- A global register allocator can do **much better** than a local allocator

Global Register Allocation: Summary

- A global register allocator can do **much better** than a local allocator
  - This justifies significant investment into solving the hard problem of global register allocation

Global Register Allocation: Summary

- A global register allocator can do **much better** than a local allocator
  - This justifies significant investment into solving the hard problem of global register allocation
- The traditional technique is **reduction to graph coloring**:

Global Register Allocation: Summary

- A global register allocator can do **much better** than a local allocator
  - This justifies significant investment into solving the hard problem of global register allocation
- The traditional technique is **reduction to graph coloring**:
  - compute **live ranges**

Global Register Allocation: Summary

- A global register allocator can do **much better** than a local allocator
  - This justifies significant investment into solving the hard problem of global register allocation
- The traditional technique is **reduction to graph coloring**:
  - compute **live ranges**
  - construct an **interference graph**

Global Register Allocation: Summary

- A global register allocator can do **much better** than a local allocator
  - This justifies significant investment into solving the hard problem of global register allocation
- The traditional technique is **reduction to graph coloring**:
  - compute **live ranges**
  - construct an **interference graph**
  - use **simplification** to color the graph, decide what to spill, and then assign physical registers

Global Register Allocation: Summary

- A global register allocator can do **much better** than a local allocator
  - This justifies significant investment into solving the hard problem of global register allocation
- The traditional technique is **reduction to graph coloring**:
  - compute **live ranges**
  - construct an **interference graph**
  - use **simplification** to color the graph, decide what to spill, and then assign physical registers
- Implementing a global register allocator correctly is a **challenge**

Global Register Allocation: Summary

- A global register allocator can do **much better** than a local allocator
  - This justifies significant investment into solving the hard problem of global register allocation
- The traditional technique is **reduction to graph coloring**:
  - compute **live ranges**
  - construct an **interference graph**
  - use **simplification** to color the graph, decide what to spill, and then assign physical registers
- Implementing a global register allocator correctly is a **challenge**
  - I don't expect all (or even most) of you to succeed at this, and it is not required for PA4

Course Announcements

- The PA4 leaderboard is still Coming Soon™
  - Unfortunately the technical problems here were tougher than I expected, so I am changing how it works
    - More details to come in the next day or two…
- Due to popular demand, I have **changed** PA4c1's specification to also allow the input to be a `.cl-type` file (originally was TAC)
  - PA4c1 is due April 28, and is mostly optional