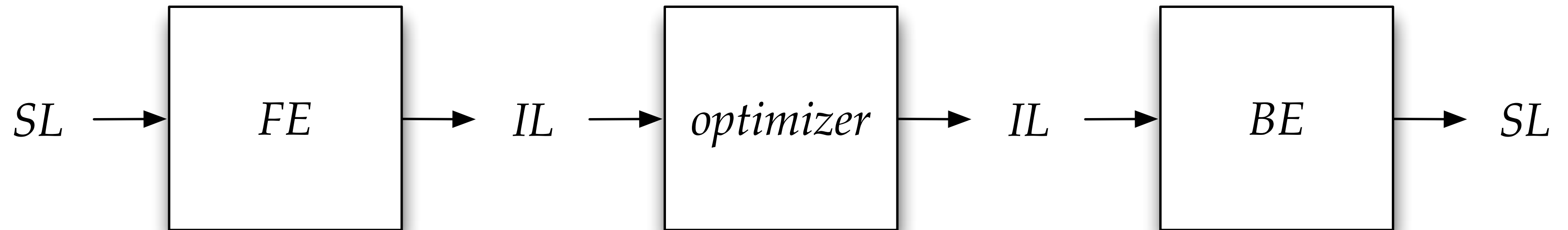
Register Allocation

Preston Briggs
Reservoir Labs

An optimizing compiler



A classical optimizing compiler (e.g., LLVM) with three parts and a nice separation of concerns:

- front end (language dependent, machine independent)
- optimizer (language independent, machine independent)
- back end (language independent, machine dependent)

Pretty optimistic!

Code generation

Pursuing our optimistic separation of concerns, we think of code generation as having three parts:

- Instruction selection,
- Register allocation, and
- Instruction scheduling.

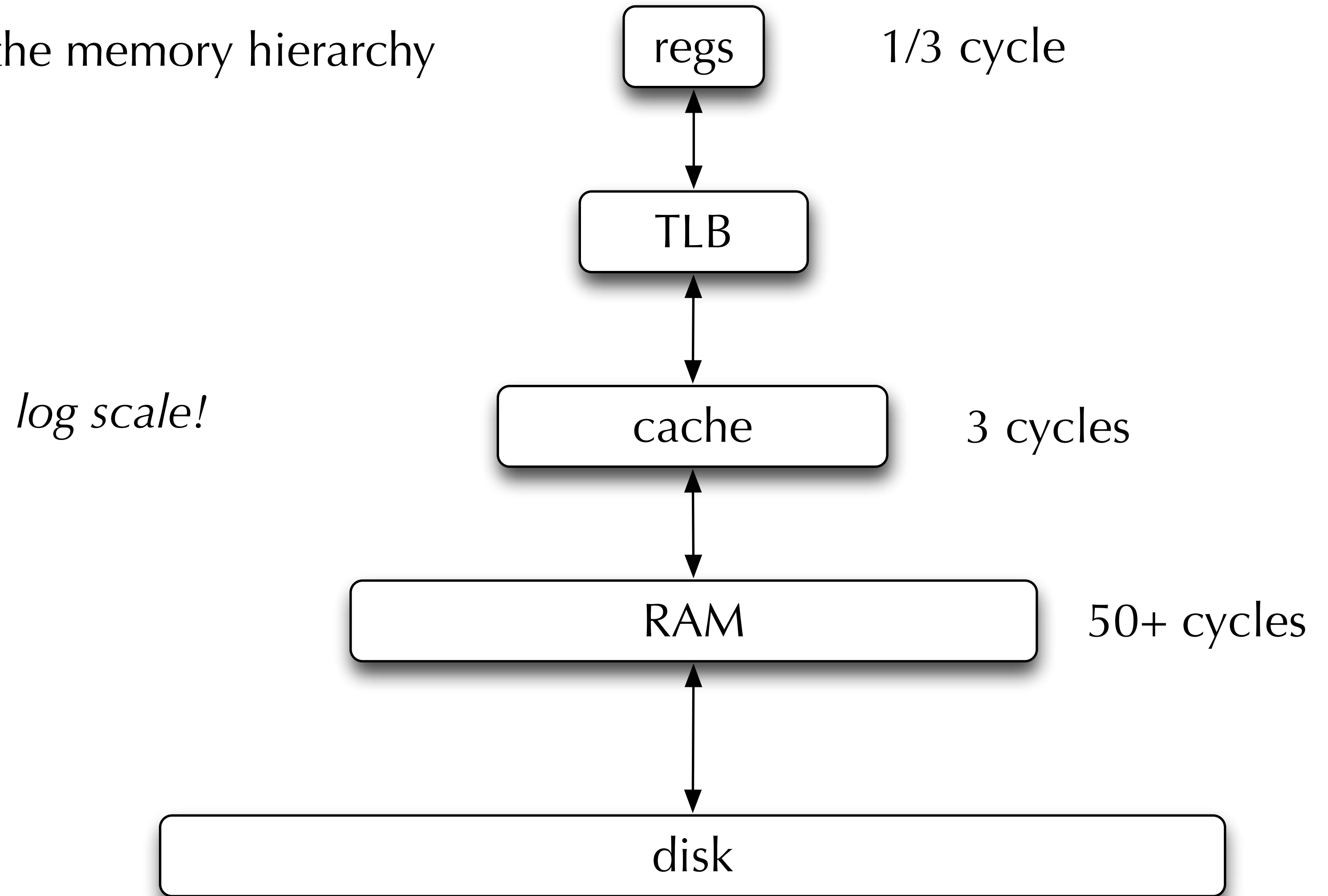
As you'll come to see, whenever there are multiple phases, there's a *phase-ordering* problem.

For some machines, one part or another won't matter to much.

In other, tougher cases, we'll have to explore new ideas.

Registers

The top of the memory hierarchy



Register allocation

As part of the overall design of his compiler, Backus suggested a simplifying separation of concerns:

> *During optimization, assume an infinite set of registers; treat register allocation as a separate problem*.

John Backus led a team at IBM building the first commercial compiler, Fortran, in 1955. He got a Turing Award.

Sheldon Best built the register allocator.

For example

Consider this simple source code:

```
int sum(int x, int y) {
  return x + y;
}
```

Here's what the IL might look like:

```
sum: enter => rX, rY
     rX + rY => r100
     return r100
```

For example, ...

Here's what the IL might look like:

```
sum: enter => r100, r101
     rX + rY => r102
     return r102
```

and here's what we might see after instruction selection (for a funny machine)

```
sum: mov   0, 100         ; copy r0 to r100
     mov   1, 101         ; copy r1 to r101
     iadd  100, 101, 102 ; r100 + r101 => r102
     mov   102, 0         ; copy r102 to r0
     rtn
```

The `mov`'s are there to satisfy the *calling convention*.

For example, ...

After instruction selection:

```
sum: mov   0, 100
     mov   1, 101
     iadd  100, 101, 102
     mov   102, 0
     rtn
```

and after register allocation

```
sum: iadd  0, 1, 0
     rtn
```

Cool!

Varieties of register allocation

Register allocation may be performed at many levels:

- Expression tree
- Local (basic block)
- Loop
- Global (routine)
- Interprocedural

Global optimization suggests global register allocation.

Interesting problems

- Control flow
- Machine details
- 2-address instructions
- Calling conventions
- Register pairs
- Restricted instructions

Finally, there are practical considerations: Space and time.

Graph coloring offers a simplifying abstraction.

Allocation via coloring

Despite C's `register` keyword, we don't allocate variables; instead, we allocate live ranges.

- A *value* corresponds to a definition
- A *live range* is composed of one or more values connected by common uses.

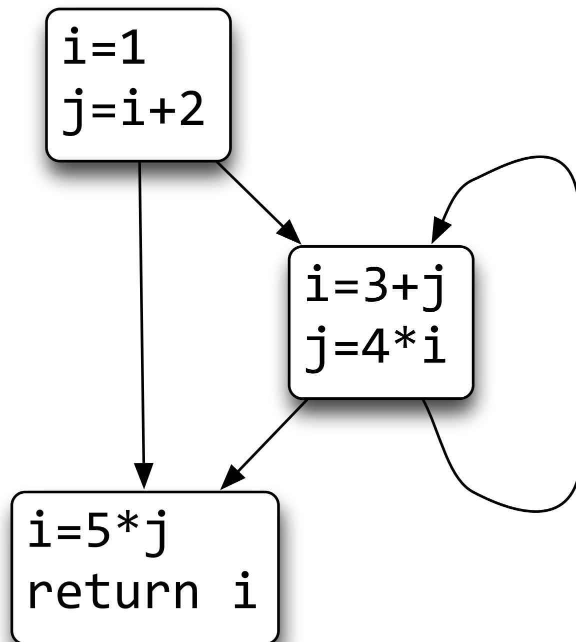
A single variable may be represented by many live ranges; furthermore, many live ranges aren't visible in the source.

We construct an *interference graph*, where

- Vertices represent live ranges
- Each edge represents an *interference* between two live ranges; i.e., both live ranges are simultaneously live and have different values.
- A k-coloring represents a register assignment.

Live ranges

Consider the live ranges in this example:



Chaitin called the process of finding live ranges *getting the right number of names*. Others called it *web analysis*. I call it *renumbering* and implement it using SSA.

Interference

Two live ranges interfere if, at some point in the routine,

- Both live ranges have been defined,
- Both live ranges will be used, and
- The live ranges have different values.

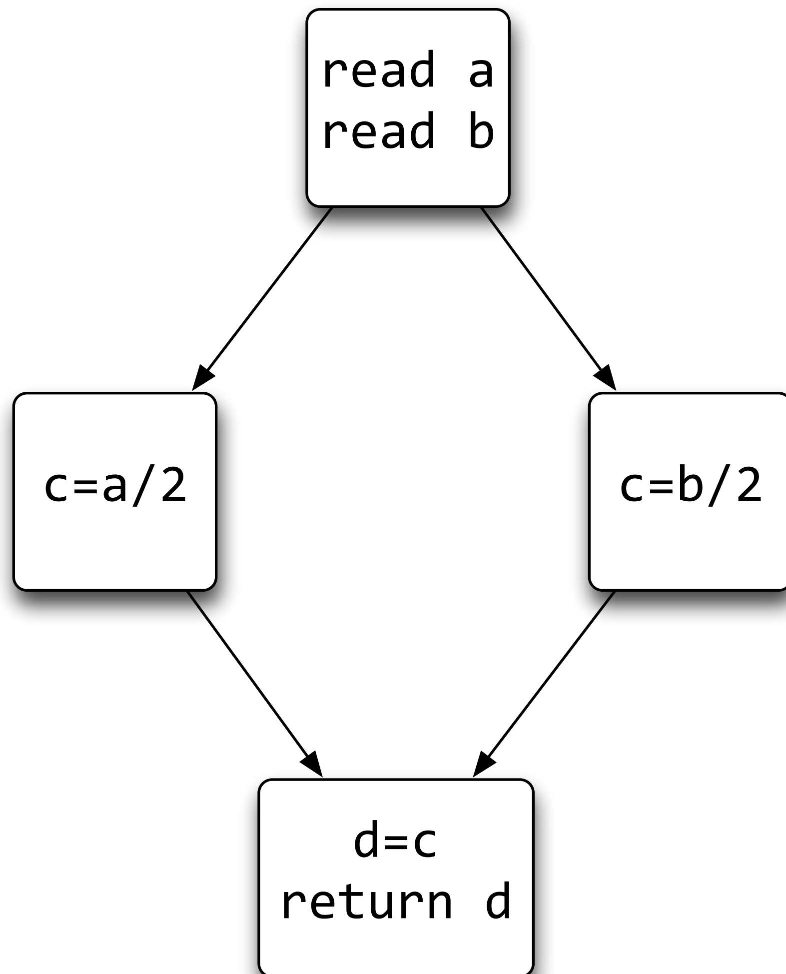
Since these conditions are undecidable, we use a conservative approximation.

At each definition in the routine, we make the defined live range interfere with all other live ranges that

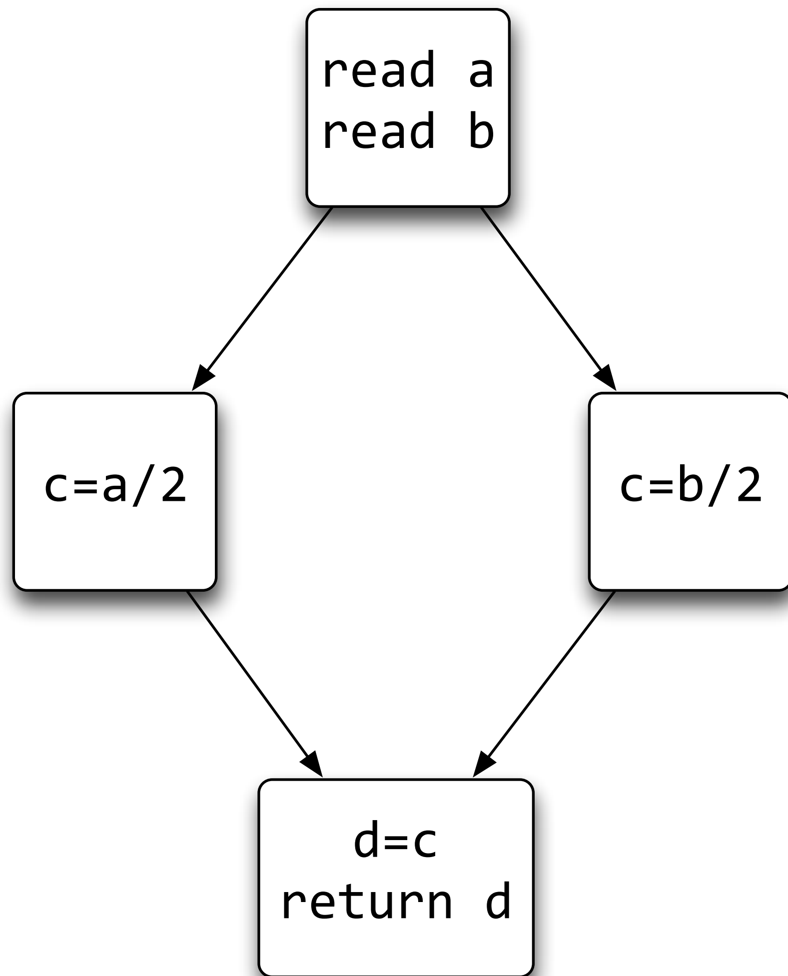
- are *live*, and
- are *available*.

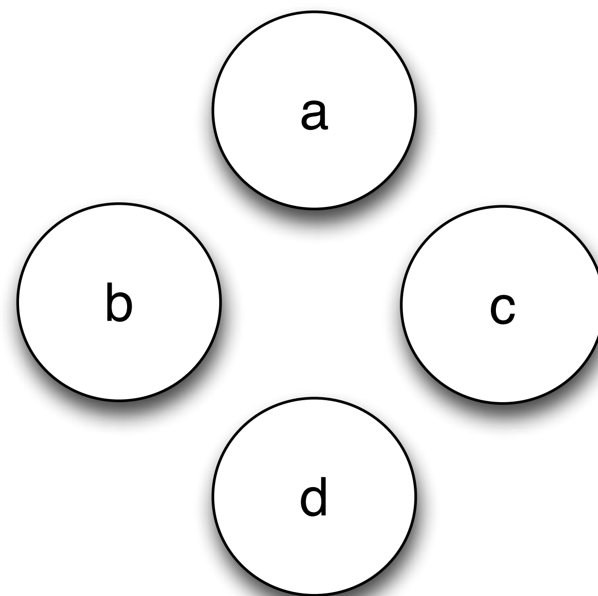
However, if the definition is a copy instruction, we don't add an interference between the source and destination edges.

For example

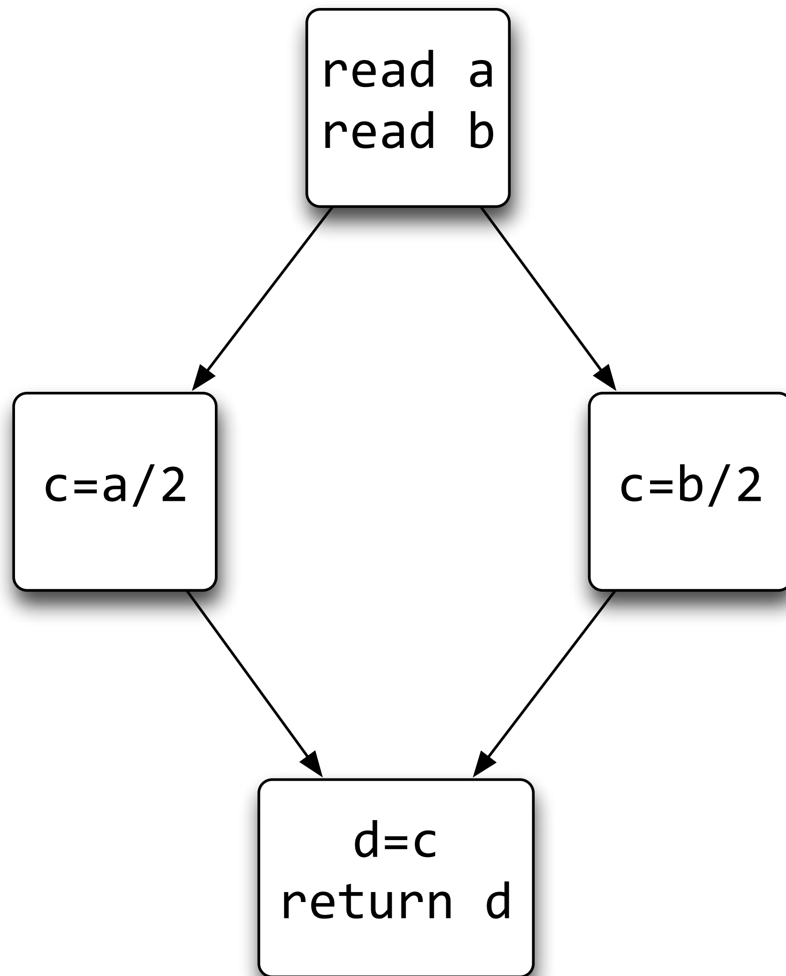


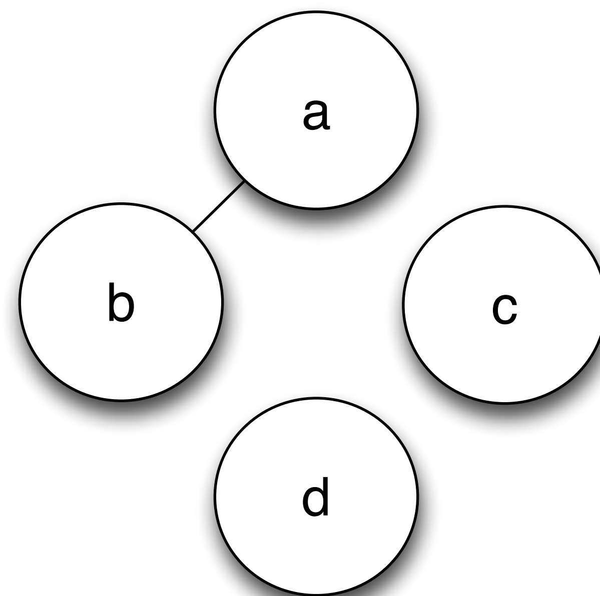
For example, ...



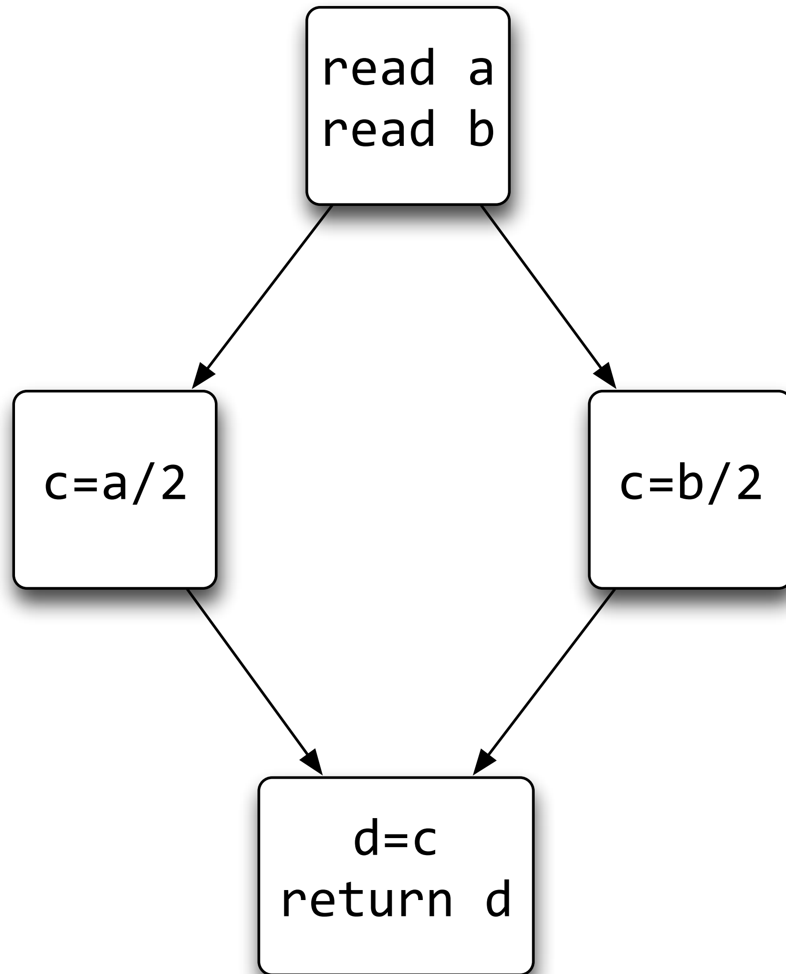


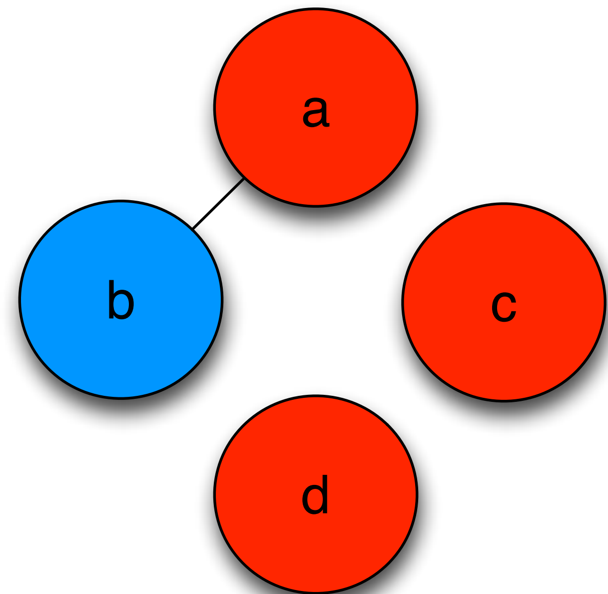
For example, ...



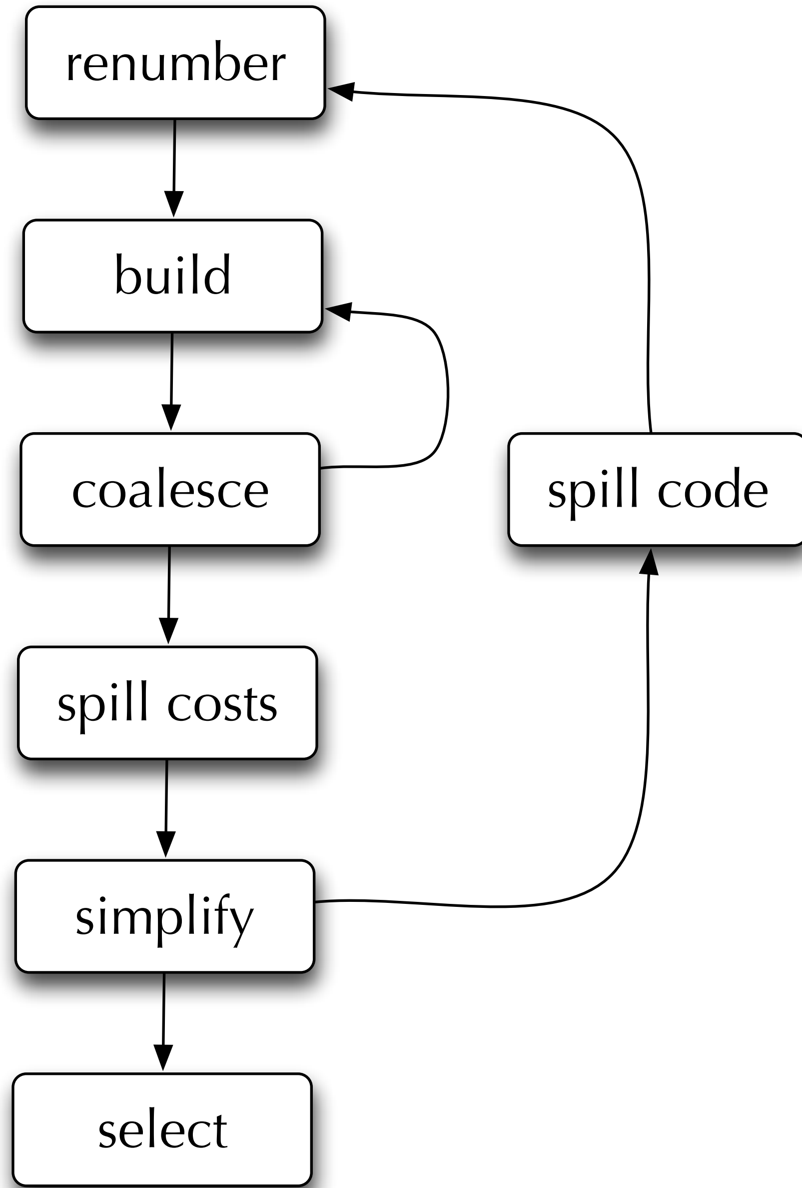


For example, ...





Chaitin's allocator



Chaitin's allocator

renumber - Find all distinct live ranges and number them uniquely.

build - Construct the interference graph.

coalesce - For each copy where the source and destination live ranges don't interfere, union the 2 live ranges and remove the copy.

spill costs - Estimate the dynamic cost of spilling each live range.

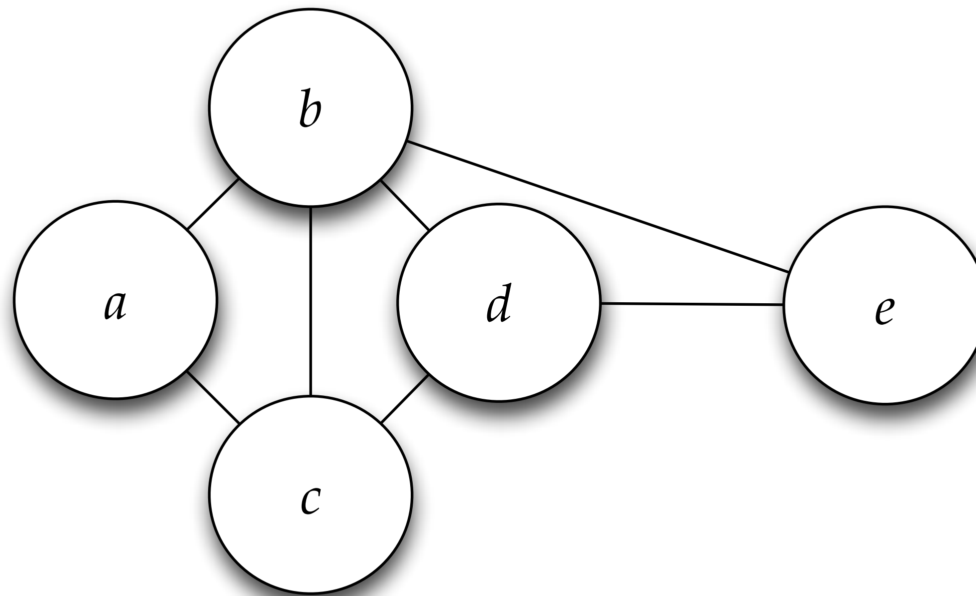
simplify - Repeatedly remove nodes with degree $< k$ from the graph and push them on a stack. If every remaining node has degree $>= k$, select a node, mark it for spilling, and remove it from the graph.

spill code - For spilled nodes, insert a load/store at each use/def and repeat from the beginning.

select - Reassemble the graph with nodes popped from the stack. As each node is added to the graph, choose a color that differs from those of the neighbors in the graph.

Another example

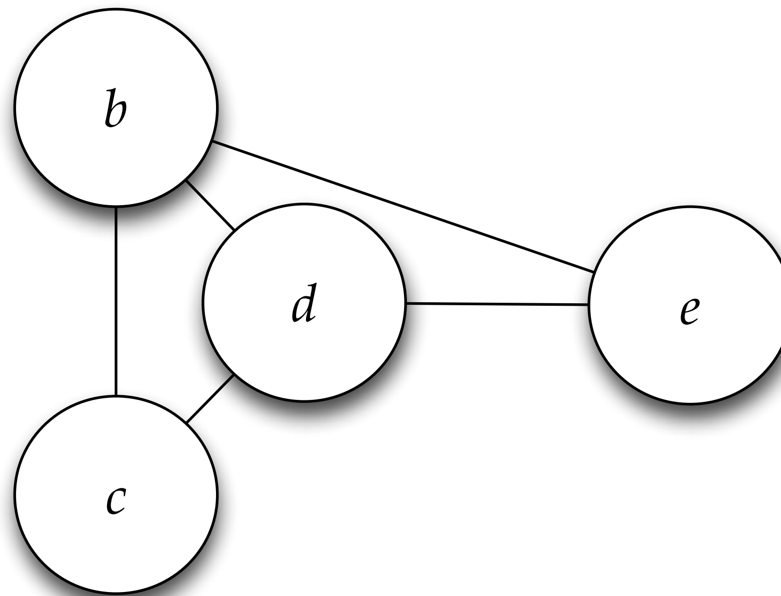
With $k = 3$



Another example, ...

With $k = 3$

a

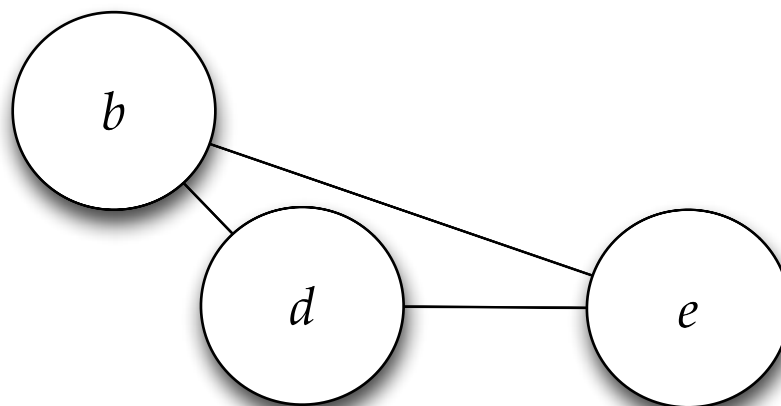


Another example, ...

With $k = 3$

a

c



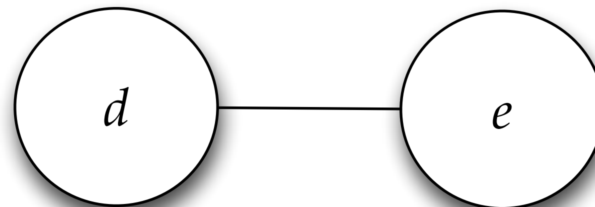
Another example, ...

With $k = 3$

a

c

b



Another example, ...

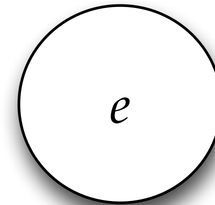
With $k = 3$

a

c

b

d



Another example, ...

With $k = 3$

a

c

b

d

e

Another example, ...

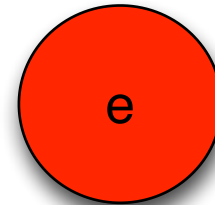
With $k = 3$

a

c

b

d



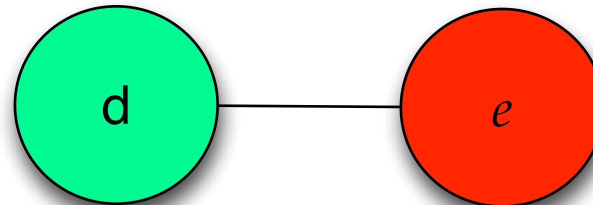
Another example, ...

With $k = 3$

a

c

b

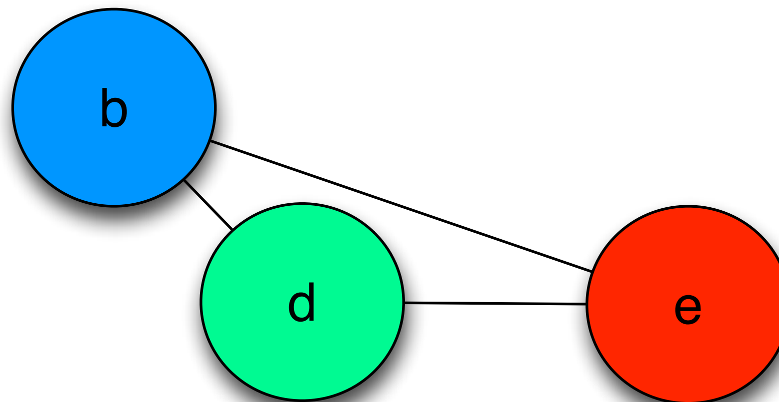


Another example, ...

With $k = 3$

a

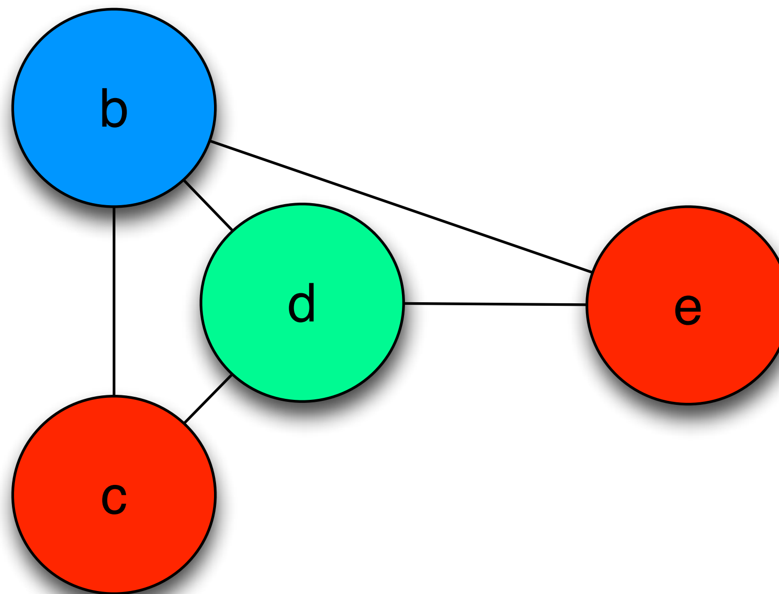
c



Another example, ...

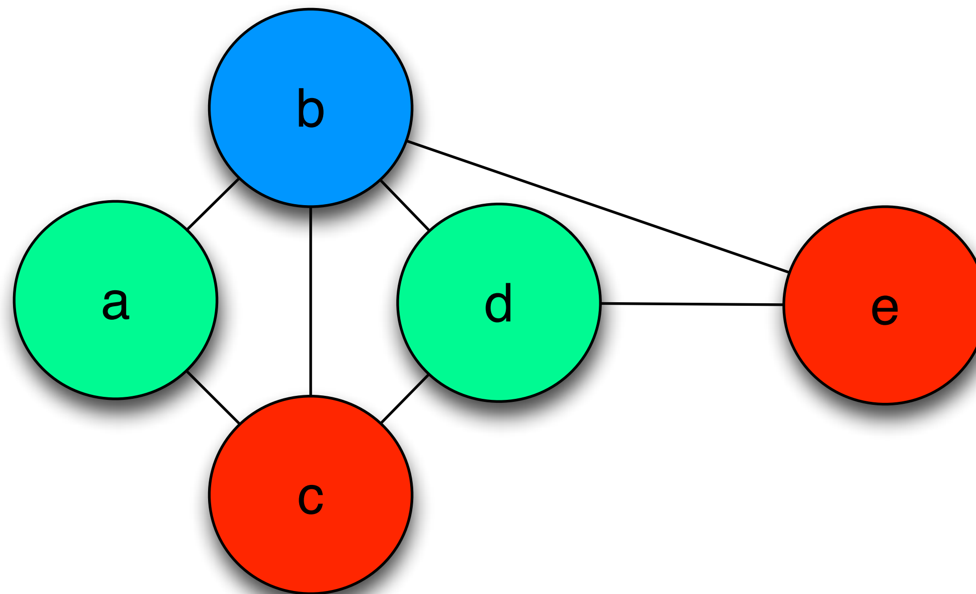
With $k = 3$

a



Another example, ...

With $k = 3$



The interference graph

The representation of the interference graph is the major factor driving space and time requirements of the allocator (and maybe even the entire compiler).

Some routines have O(5K) nodes and O(1M) edges.

Required operations are:

new(*n*) - Return a new graph with *n* nodes, but no edges

add(*g, x, y*) - Return a graph including *g* and an edge between *x* and *y*

interfere(*g, x, y*) - Return true if there's an edge between *x* and *y* in *g*

degree(*g, x*) - Return the number of neighbors of *x* in *g*

neighbors(*g, x*) - Return the set of neighbors of *x* in *g*

The interference graph, ...

Chaitin used a dual representation

- A triangular bit matrix, supporting efficient random access, and
- A vector of adjacency vectors, supporting efficient access to the neighbors of a node.

The structures are initialized in two passes over the code.

- Before the 1st pass, allocate and clear the bit matrix. During the 1st pass, fill in the matrix and accumulate the number of neighbors for each node.
- Before the 2nd pass, allocate the adjacency vectors and clear the bit matrix. During the 2nd pass, rebuild the bit matrix, adding entries to the adjacency vectors.
- Since coalescing corrupts the graph, the 2nd pass is repeated until all coalescing is complete.
- After coalescing, space for the bit matrix may be reclaimed.

Spilling

Generally the spill cost of a live range is the weighted sum of the number of uses and defs, where each use and def is weighted proportionally to its loop-nesting depth.

However, this neglects two important refinements introduced by Chaitin:

1. There's no benefit in spilling a live range between 2 uses or between a def and a use if no other live ranges die in the interval.
2. Some live ranges should be rematerialized instead of being spilled to memory.

For best results, these details should play into the computation of spill costs as well as creating spill code.

Chaitin's contribution

The 1st complete allocator based on graph coloring was descibed by Chaitin, et al. in 1981. They developed

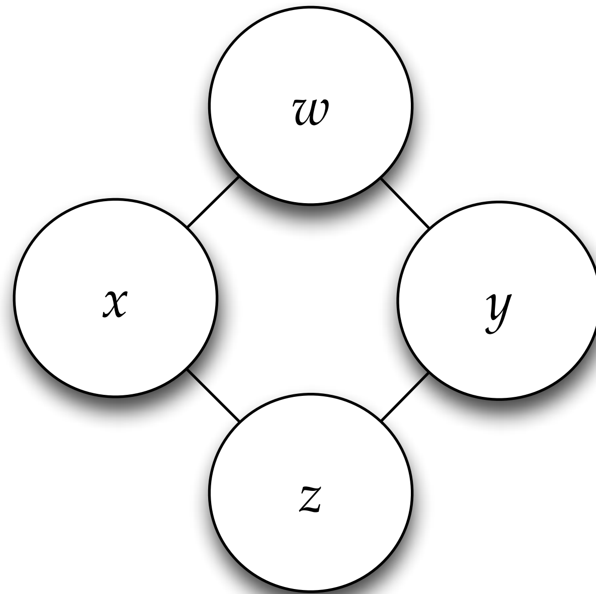
- a precise notion of *interference*,
- *coalescing*,
- an efficient coloring heuristic, and
- efficient data structures and algorithms for managing the interference graph.

Additionally, they showed how to manipulate the interference graph in a systematic fashion to account for many common machine "features."

Chaitin's '82 paper gives a coloring heuristic that extends naturally to handle spilling.

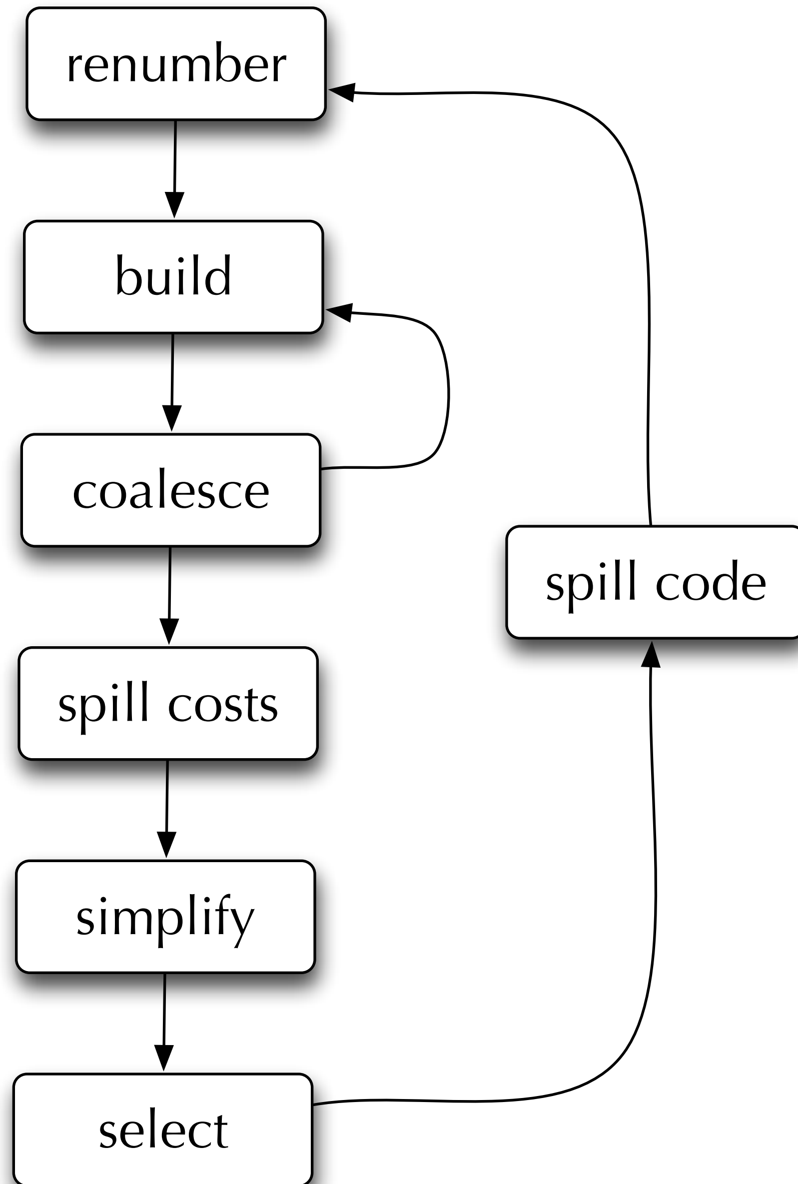
A problem

Ken Kennedy showed me this counter example:



Clearly there's a 2-coloring, but Chaitin's heuristic won't find it.

The optimistic allocator



Instead of spilling, Briggs pushes the spill candidate on the stack, hoping there will be a color available.

All nodes go on the stack.

By deferring spill decisions, Briggs wins twice:

- when neighbors of a node get the same color, and
- when a neighbor has already been spilled.

Chaitin and Briggs

- Chaitin's method colors a subset of the graphs Briggs colors.
- Any live range Briggs spills, Chaitin spills.
- Chaitin often spills more live ranges.
- Improvements can be significant.
- Briggs achieves the same time bounds as Chaitin.

Realistically, Chaitin did all of the hard work;
Briggs made a small improvement and drew pretty figures.