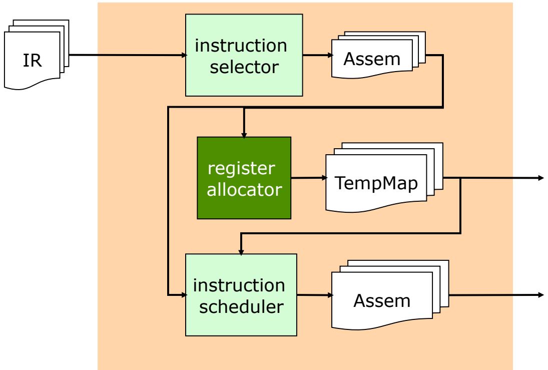
Register Allocation

15-745 Optimizing Compilers
Spring 2008

Carnegie Mellon
School of Computer Science

Back end structure



Carnegie Mellon
School of Computer Science

After Instruction Selection

```
define i32 @erk(i32 %a, i32 %b) {
entry:
        %tmp3 = mul i32 %a, 37                  ; <i32> [#uses=1]
        %tmp4 = mul i32 %tmp3, %a               ; <i32> [#uses=1]
        %tmp6 = add i32 %b, 33                  ; <i32> [#uses=1]
        %tmp7 = sdiv i32 %tmp4, %tmp6           ; <i32> [#uses=1]
        %tmp9.neg = xor i32 %a, -1              ; <i32> [#uses=1]
        %tmp10 = add i32 %tmp7, %tmp9.neg                 ; <i32> [#uses=1]
        ret i32 %tmp10
}
```

```
entry: 0x803ce0, LLVM BB @0x801a90, ID#0:
        %reg1024 = MOV32ri 33
        %reg1025 = ADD32rm %reg1024, <fi#-2>, 1, %NOREG, 0
        %reg1026 = MOV32rm <fi#-1>, 1, %NOREG, 0
        %reg1027 = IMUL32rr %reg1026, %reg1026
        %reg1028 = IMUL32rri8 %reg1027, 37
        %EAX = MOV32rr %reg1028
        %EAX = CDQ %EDX<imp-def>, %EAX<imp-use>
        IDIV32r %reg1025, %EAX<imp-def>, %EDX<imp-def>, %EAX<imp-use>, %EDX<imp-use>
        %reg1029 = MOV32rr %EAX
        %reg1030 = NOT32r %reg1026
        %reg1031 = ADD32rr %reg1029, %reg1030
        %EAX = MOV32rr %reg1031
        RET
```

Carnegie Mellon
School of Computer Science

Abstract View

```
…
movl   $2, t7
movl   t7, t11
imull  t7, t11
movl   t11, t10
imull  $37, t10
movl   t10, t8
movl   t7, t17
addl   $1, t17
movl   $33, %eax
cltd
idivl  t17
…
```

```
Written         Read
…
t7        <- ()
t11       <- (t7)
t11       <- (t7, t11)
t10       <- (t11)
t10       <- (t10)
t8        <- (t10)
t17       <- (t7)
t17       <- (t17)
%eax      <- ()
%edx,%eax <- (%eax)
%eax,%edx <- (t17,%eax,%edx)
…
```

Abstract view, for register allocation purposes

Register allocator's job

The register allocator's job is to assign each temp to a machine register.

If that fails, the register allocator must rewrite the code so that it can succeed.

```
…
t7   <- 2
t11  <- t7
t11  <- (t7, t11)
t10  <- t11
t10  <- t10
t8   <- t10
t17  <- t7
t17  <- t17
%eax <-
%edx <- (%eax,%edx)
%eax,%edx <- t17
…
```

```
…
t7  : %ebx
t8  : %ecx
t10 : %eax
t11 : %eax
t17 : %esi
…
```

The TempMap

Some terminology

Two temps *interfere* if at some point in the program they cannot both occupy the same register.

Which temps interfere?

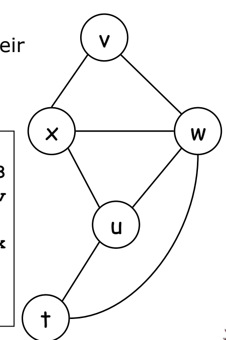
```
v <-  1
w <-  v + 3
x <-  w + v
u <-  v
t <-  u + x
  <-  w
  <-  t
  <-  u
```


A graph-coloring problem

Interference graph: an undirected graph where

- nodes = temps
- there is an edge between two nodes if their corresponding temps interfere

```
v <-  1
w <-  v + 3
x <-  w + v
u <-  v
t <-  u + x
  <-  w
  <-  t
  <-  u
```

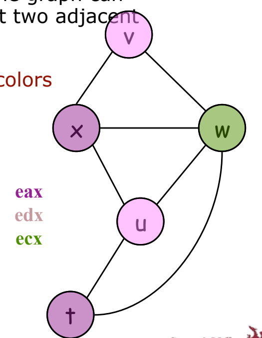


A graph-coloring problem

A graph is *k-colorable* if every node in the graph can be colored with one of k colors such that two adjacent nodes do not have the same color

Assigning k registers = Coloring with k colors

```
v <-  1
w <-  v + 3
x <-  w + v
u <-  v
t <-  u + x
  <-  w
  <-  t
  <-  u
```



History

For early architectures, register allocation was not very important

Early work by Cocke (in 1971) proposed the idea that register allocation can be viewed as a graph coloring problem

Chaitin was the first to implement this idea for the IBM 370 PL/1 compiler, in 1981

In 1982, at IBM, Chaitin's allocator was used for the PL.8 compiler for the IBM 801 RISC system

Today, register allocation is the most essential of code optimizations

History, cont'd

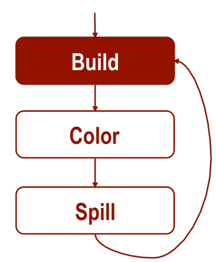
Motivated by the first MIPS architecture, Chow and Hennessy developed priority-based graph coloring in 1984

"top down" coloring

Another popular algorithm for register allocation based on graph coloring is due to Briggs in 1992

"bottom up" coloring

Steps in register allocation

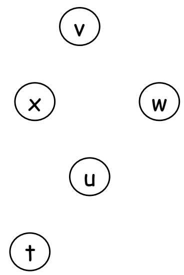


Building the interference graph

Given liveness information, we can build the interference graph (IG)

```
v <-  1     | {}
w <-  v + 3 | {v}
x <-  w + v | {w,v}
u <-  v     | {w,x,v}
t <-  u + x | {w,u,x}
  <-  x     | {x,w,u,t}
  <-  w     | {w,u,t}
  <-  t     | {u,t}
  <-  u     | {u}
```

How?



Edges of Interference Graph

Intuitively:

Two variables interfere if they overlap at some point in the program.

Algorithm:

At each point in program,
enter an edge for every pair of variables at that point

An optimized definition & algorithm for edges:

```
For each defining inst i
     Let x be definition at inst i
     For each variable y live at end of inst i
          insert an edge between x and y
```

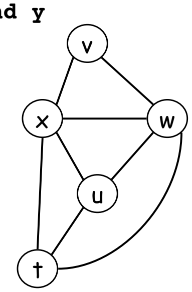
Faster?

Better quality? $\boxed{A = 2 \ (A_2)}$ {D} {A,D} ⟶ Edge between A and D

Building the interference graph

```
for each defining inst i
  let x be temp defined at inst i
  for all y in LIVE-IN of succ(i)
    insert an edge between x and y
```

```
v <-  1     | {}
w <-  v + 3 | {v}
x <-  w + v | {w,v}
u <-  v     | {w,x,v}
t <-  u + x | {w,u,x}
  <-  x     | {x,w,u,t}
  <-  w     | {w,u,t}
  <-  t     | {u,t}
  <-  u     | {u}
```



A Better Interference Graph

```
x = 0;
for(i = 0; i < 10; i++)
{
   x += i;
}

y = global;
y *= x;

for(i = 0; i < 10; i++)
{
   y += i;
}
```

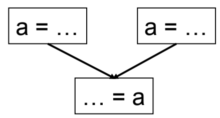
What does the interference graph look like?

What's the minimum number of registers needed?

Live Ranges & Merged Live Ranges

A *live range* consists of a definition and all the points in a program (e.g. end of an instruction) in which that definition is live.

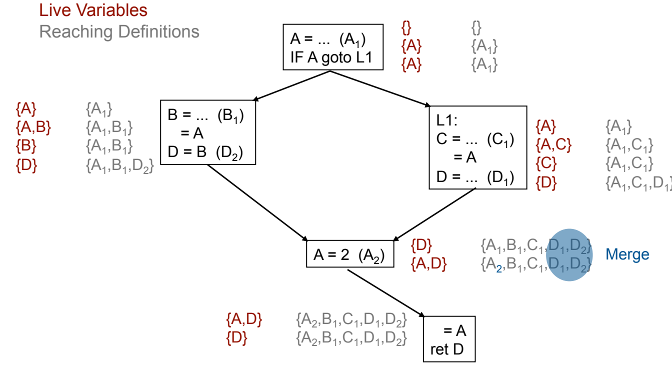
– How to compute a live range?



Two overlapping live ranges for the same variable must be merged

Example

Live Variables
Reaching Definitions



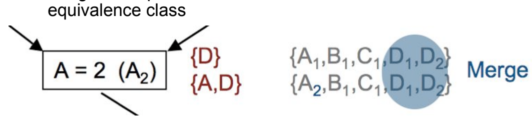
A *live range* consists of a definition and all the points in a program in which that definition is live.

Carnegie Mellon
School of Computer Science

Merging Live Ranges

Merging definitions into equivalence classes:

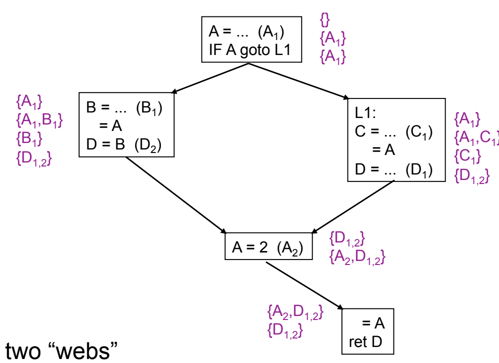
- Start by putting each definition in a different equivalence class
- For each point in a program
  - if variable is live, and there are multiple reaching definitions for the variable
  - merge the equivalence classes of all such definitions into a one equivalence class



Merged live ranges are also known as “webs”

Carnegie Mellon
School of Computer Science

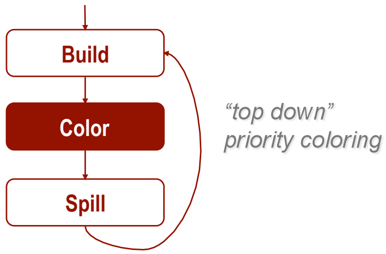
Example: Merged Live Ranges



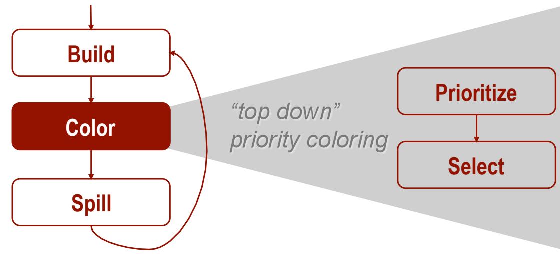
A has two “webs”
makes register allocation easier

Carnegie Mellon
School of Computer Science

Steps in register allocation



Steps in register allocation



Priority Coloring

Heuristic priority function computes priority for each variable

- highest priority allocated first

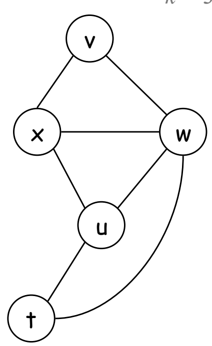
Ideas for priority functions?

k = 3

```
v <-  1
w <-  v + 3
x <-  w + v
u <-  v
t <-  u + x
  <-  w
  <-  t
  <-  u
```

Priorities:
v: 4
w: 3
x: 2
u: 3
t: 2

Order:
v, w, u, x, t



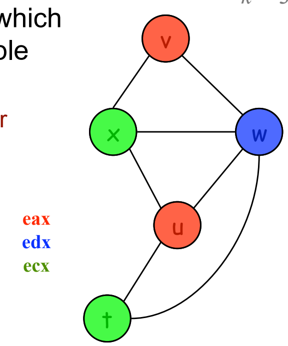
Priority Coloring

Allocate in priority order

Another heuristic selects which register to assign to variable

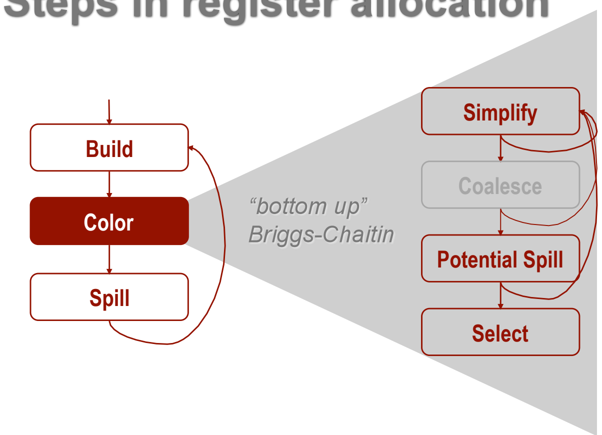
- rotating registers
- lowest numbered register

k = 3



Order:
v, w, u, x, t

Steps in register allocation



Graph coloring

Once we have the interference graph, we can attempt register allocation by searching for a K-coloring

This is an NP-complete problem (K≥3)*

But a linear-time simplification algorithm by Kempe (in 1879) tends to work well in practice

* [1] H. Bodlaender, J. Gustedt, and J. A. Telle, "Linear-time register allocation for a fixed number of registers," in *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, pp. 574–583, Society for Industrial and Applied Mathematics, 1998.
[2] S. Kannan and T. Proebsting, "Register allocation in structured programs," in *Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pp. 360–368, Society for Industrial and Applied Mathematics, 1995.
[3] M. Thorup, "All structured programs have small tree width and good register allocation," *Inf. Comput.*, vol. 142, no. 2, pp. 159–181, 1998.

Carnegie Mellon
School of Computer Science

Kempe's algorithm

Basic observation:

- given a graph that contains a node with degree less than K, the graph is K-colorable iff the graph without that node is K-colorable
- this is called the "degree<K" rule

So, step #1 of Kempe's algorithm:

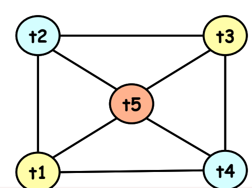
- iteratively remove nodes with degree<K

Carnegie Mellon
School of Computer Science

Kempe's algorithm, cont'd

If all nodes are removed by step #1, then the graph is K-colorable

However, the degree<K rule does not always work, for example:



This graph is 3-colorable, but the degree<3 rule doesn't work

Carnegie Mellon
School of Computer Science

Kempe's algorithm, cont'd

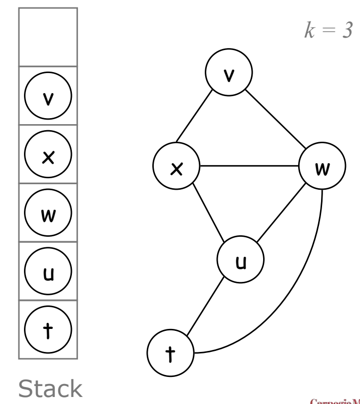
In step #1, each removed node should be pushed onto a stack

- when all are removed, we pop each node and put it back into the graph, assigning a suitable color as we go

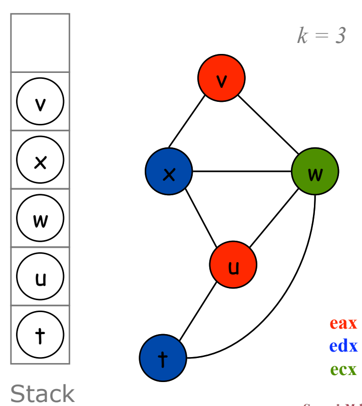
In case we get stuck (i.e., there are no nodes with degree<K), we apply step #2:

- choose a node with degree≥K and optimistically remove it, and then continue

Example



Example



Another Example

Now what?

Be optimistic:

- Put a node with degree ≥ k on stack

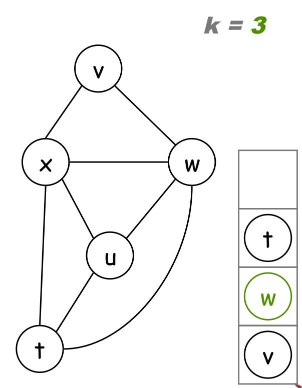
- Lose guarantee that anything we put on stack is colorable

- If we're lucky this node will still be colorable when popped from stack

Be realistic:

- If unlucky, this node will have to be spilled (allocated to memory)

- Mark as *potential spill* to avoid recomputation later



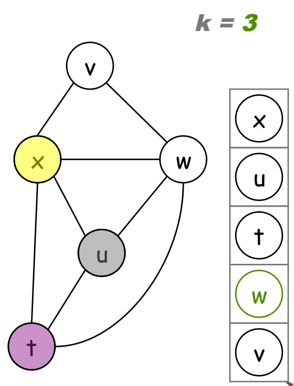
Select

Pop a node from the stack

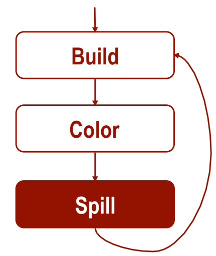
Assign it a color that does not conflict with neighbors in interference graph

This will always be possible, *unless* the node is a potential spill

If it is not possible, mark as actual spill



Steps in register allocation



Spilling to Memory

RISC Architectures

- Only load and store can access memory
  - every use requires load
  - every def requires store
  - create new temporary for each location

CISC Architectures

- can operate on data in memory directly
  - makes writing compiler easier(?), but isn't necessarily faster
- pseudo-registers inside memory operands still have to be handled

Spilling a use

For an instruction like
- t <- (u,v)

If u is marked as an actual spill, transform to
- u' := **u** *(i.e., a load instruction)*
- t <- (u',v)

where u' is a new temp

u and u' are special:
- **u** is spilled and thus *unallocatable*
- u' is marked as *unspillable*

Spilling a def

For an instruction like
- t <- (u,v)

If t is marked as an actual spill, transform to
- t' <- (u,v)
- **t** := t' *(i.e., a store instruction)*

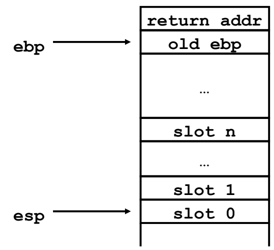
where t' is a new temp

t and t' are special:
- **t** is spilled and thus *unallocatable*
- t' is marked as *unspillable*

Spilled (unallocable) temps

Question: Where do the spilled temps get stored?

Answer: On the stack, in *stack slots*



Each spilled temp should be allocated into a stack slot

The compiler can maintain a counter for the "next" slot number

To "mark" an actual spill, give it a slot number

Carnegie Mellon School of Computer Science

Stack slots

In order to create the stack slots at run time, the prelude code needs to modify %esp

```
_main:
      pushl  %ebp
      movl   %esp, %ebp
      movl   %edi, t2
      movl   %ebx, t3
      movl   %esi, t4
```

```
_main:
      pushl  %ebp
      movl   %esp, %ebp
      movl   %edi, t2
      movl   %ebx, t3
      movl   %esi, t4
      subl   $(n×4), %esp
```

Note that the subl can be generated only after register allocation is finished

Carnegie Mellon School of Computer Science

Spill code generation

The effect of spill code generation is to turn long live ranges into a bunch of tiny live ranges

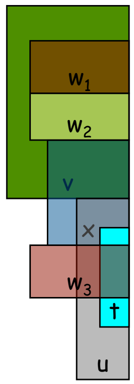
This introduces new temps

Hence, register allocation must start over from scratch whenever spill code is generated

Carnegie Mellon School of Computer Science

Spilling

```
v  <-   1
w1 <-   v + 3
Mw[]<-  w1
w2 <-   Mw[]
x  <-   w2 + v
u  <-   v
t  <-   u + x
   <-   x
w3 <-   Mw[]
   <-   w3
   <-   t
   <-   u
```



Allocate w to memory location M_w

Spilled variables are allocated to the stack in an area completely controlled by the compiler. These memory locations are special in that they can be optimized without concern for memory aliasing issues.

Now Start Over...

...compute live ranges...

What to Spill?

When choosing potential spill node want:

- A node that makes graph easier to color
  - Fewer spills later
- A node that isn't "expensive" to spill
  - An expensive node would slow down the program if spilled
- We can apply heuristics both when choosing potential spill nodes and when choosing actual spill nodes
  - not required to spill node that we popped off stack and can't color

Carnegie Mellon School of Computer Science

A Spill Heuristic

Pick node (live range) *n* that minimizes:

$$\frac{\sum_{def \in n} 10^{depth(def)} + \sum_{use \in n} 10^{depth(use)}}{\text{degree}(n)}$$

This heuristic prefers nodes that:

- Are used infrequently
- Aren't used inside of loops
- Have a large degree

Could use any one of several other heuristics as well…

Carnegie Mellon School of Computer Science

Rematerialization

An alternative to spilling

- Recompute value of variable instead of store/load to memory
- Example:

```
v <-  1
w <-  v + 3
x <-  w + v
u <-  v
t <-  u + x
  <-  w
  <-  t
  <-  u
```

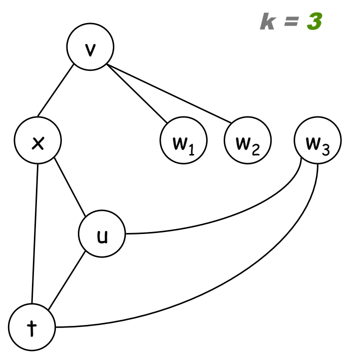
→

```
v <-  1
w <-  v + 3
x <-  w + v
u <-  v
t <-  u + x
w <-  4
  <-  w
  <-  t
  <-  u
```

Carnegie Mellon School of Computer Science

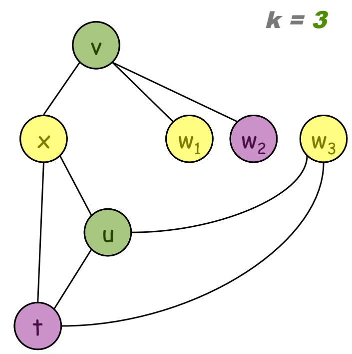
Build Take Two

```
v  <-   1
w1 <-   v + 3
Mw[]<-  w1
w2 <-   Mw[]
x  <-   w2 + v
u  <-   v
t  <-   u + x
   <-   x
w3 <-   Mw[]
   <-   w3
   <-   t
   <-   u
```



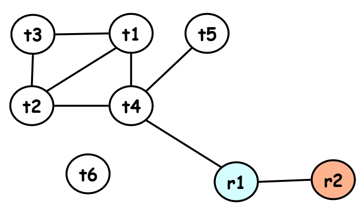
Recalculate interference graph

Simplify->Select



Another Example

```
t1 <- ()
t2 <- ()
t3 <- (t1,t2)
t4 <- (t1,t3)
t5 <- (t1,t2)
t6 <- (t4,t5)
```

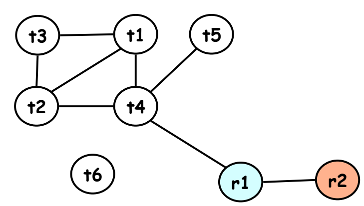


Assume 2 machine registers, {r1,r2}.

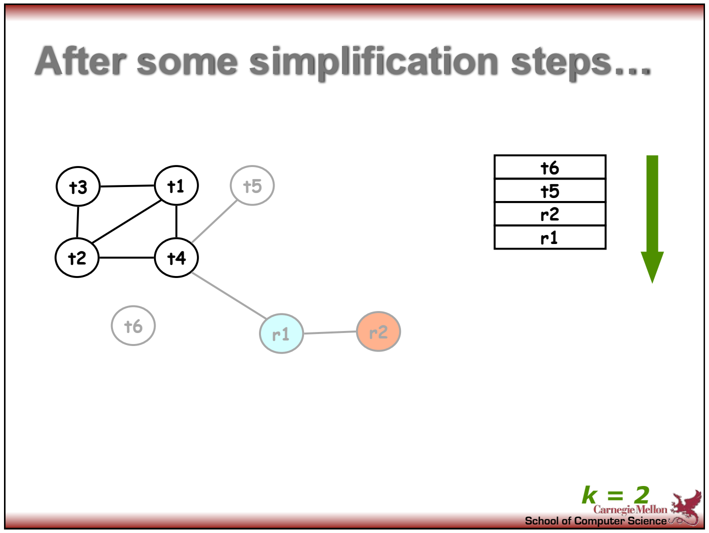
Assume t4 may not be in r1.

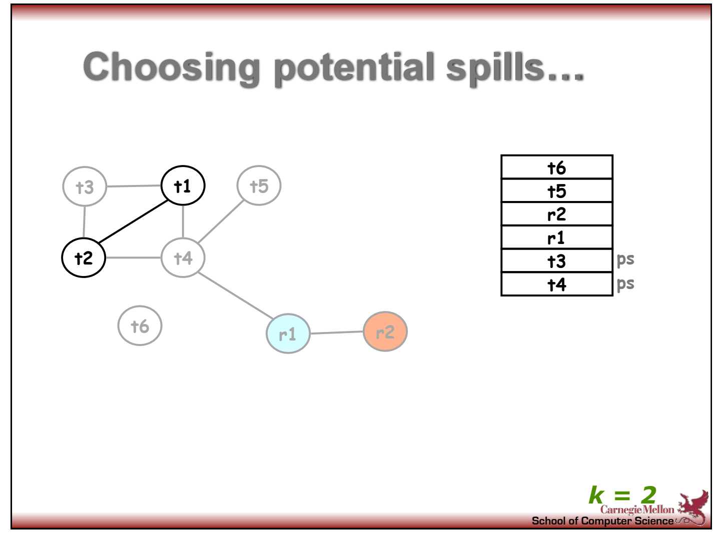
Then we have the ***interference graph***

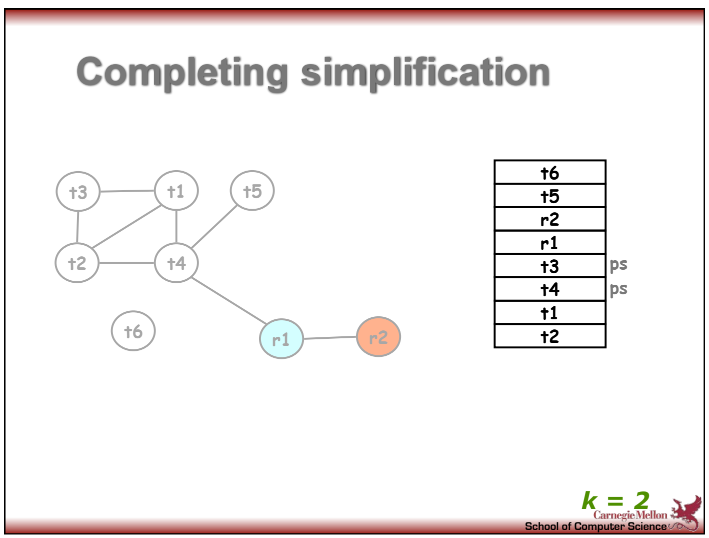
Simplification steps...

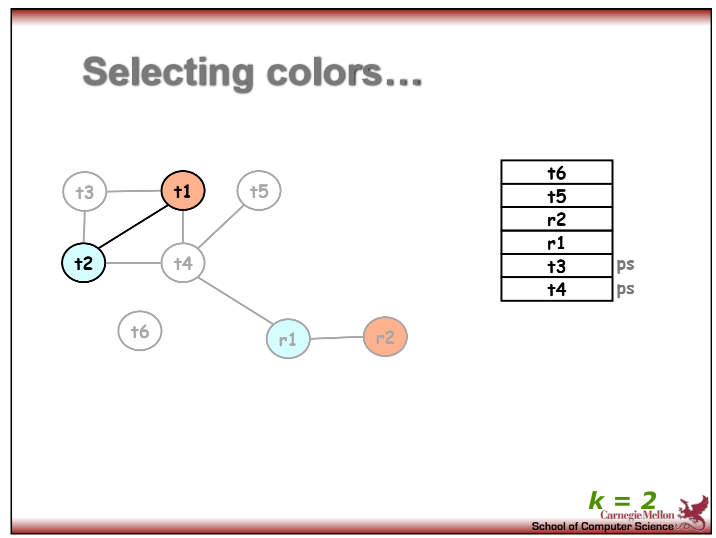


k = 2

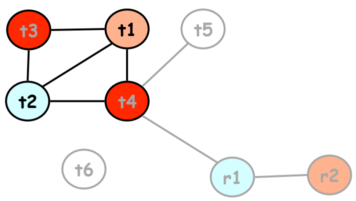

After some simplification steps…
t3
t1
t5
t2
t4
t6
r1
r2
t6
t5
r2
r1
k = 2
Carnegie Mellon
School of Computer Science


Choosing potential spills…
t3
t1
t5
t2
t4
t6
r1
r2
t6
t5
r2
r1
t3 ps
t4 ps
k = 2
Carnegie Mellon
School of Computer Science


Completing simplification
t3
t1
t5
t2
t4
t6
r1
r2
t6
t5
r2
r1
t3 ps
t4 ps
t1
t2
k = 2
Carnegie Mellon
School of Computer Science

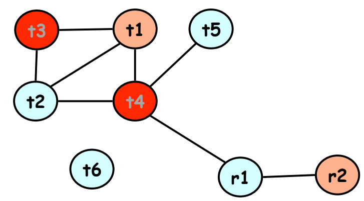

Selecting colors…
t3
t1
t5
t2
t4
t6
r1
r2
t6
t5
r2
r1
t3 ps
t4 ps
k = 2
Carnegie Mellon
School of Computer Science

Actual spills…

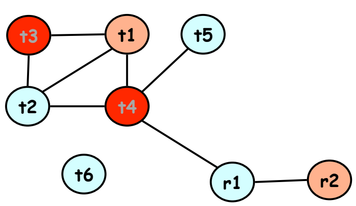


t6
t5
r2
r1

Select complete!



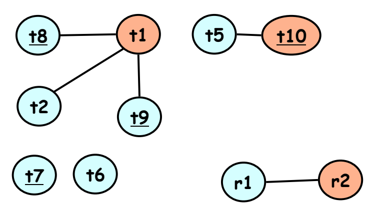
Spill code generation…




```
t1 <- ()
t2 <- ()
t7 <- (t1,t2)
t3  := t7
t8  := t3
t9 <- (t1,t8)
t4  := t9
t5 <- (t1,t2)
t10  := t4
t6 <- (t10,t5)
```

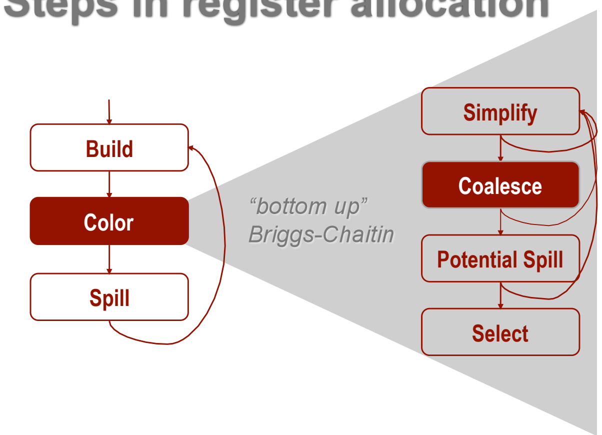
Notice: Live ranges for t3 and t4 have been chopped up into lots of small live ranges

…and start over!




```
r2 <- ()
r1 <- ()
r1 <- (r2,r1)
slot0  := r1
r1  := slot0
r1 <- (r2,r1)
slot1  := r1
r1 <- (r2,r1)
r2  := slot1
r1 <- (r2,r1)
```

Steps in register allocation



Move Coalescing

Eliminate moves by assigning the src and dest to the same register

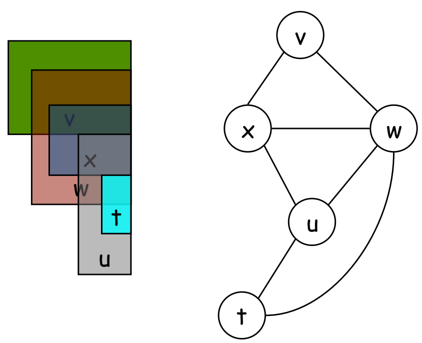
```
movl t1,t2    ➡  movl %eax,%eax   ➡  addl %edx,%eax
addl t3,t2       addl %edx,%eax
```

When can we coalesce t1 and t2?

How can we modify our interference graph to do this?

Example

```
v <- 1
w <- v + 3
x <- w + v
u <- v
t <- u + x
  <- w
  <- t
  <- u
```



First compute live ranges...

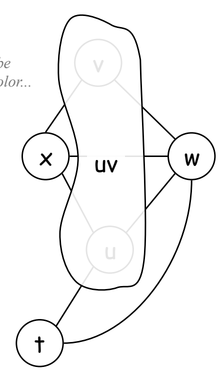
...then construct interference graph

Example

```
v <- 1
w <- v + 3
x <- w + v
u <- v
t <- u + x
  <- w
  <- t
  <- u
```

Want u and v to be assigned same color...

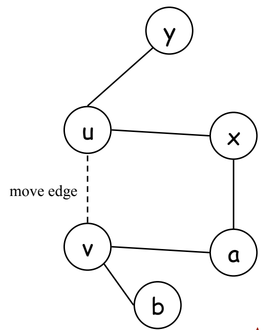
...merge u and v to form a single node



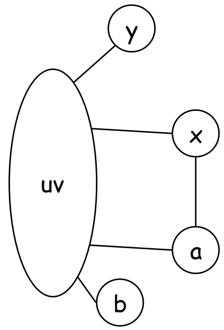
u and v are special:
A move whose source is not live-out of the move is a candidate for coalescing

That is, if the src and dest don't interfere

Is Coalescing Always Good?



vs.



And the winner is?

2 colorable

3 colorable

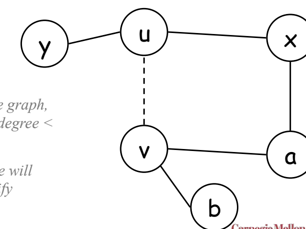
When should we coalesce?

Always
- If we run into trouble start un-coalescing
  - no nodes with degree < k, see if breaking up coalesced nodes fixes
- Can we always coalesce **all** coalescable moves?

Only if we can prove it won't cause problems
- Briggs: Conservative Coalescing
- George: Iterated Coalescing

When we simplify the graph, we remove nodes of degree < k...

want to make sure we will still be able to simplify coalesced node, uv



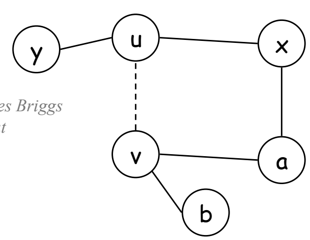
Briggs: Conservative Coalescing

- Can coalesce u and v if:
  - (# of neighbors of uv with degree ≥ k) < k
- Why?
  - *Simplify* pass removes all nodes with degree < k
  - # of remaining nodes < k
  - Thus, uv can be simplified

What does Briggs say about

k = 3?

k = 2?



George: Iterated Coalescing

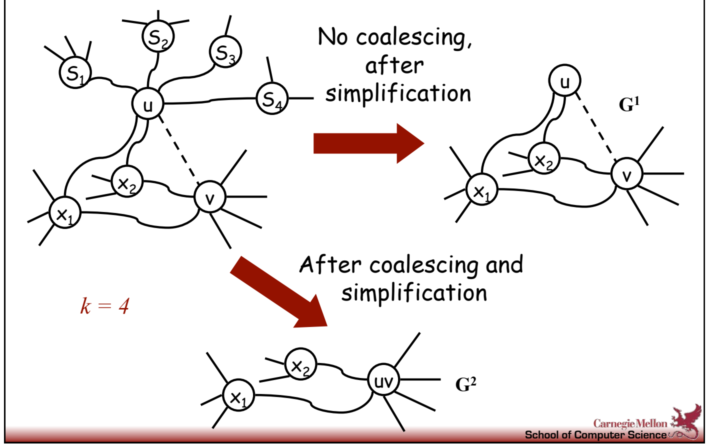
Can coalesce u and v if

foreach neighbor t of u
- t interferes with v, or, *doesn't change degree*
- degree of t < k *removed by simplification*

Resulting node uv will (after simplification) have degree equal to degree of v

Why?
- let S be set of neighbors of u with degree < k
- If no coalescing, simplify removes all nodes in S, call that graph G^1
- If we coalesce we can still remove all nodes in S, call that graph G^2
- G^2 is a subgraph of G^1

George: Iterated Coalescing



Why Two Methods?

•Why not?
•With Briggs, one needs to look at all neighbors of a & b
•With George, only need to look at neighbors of a.

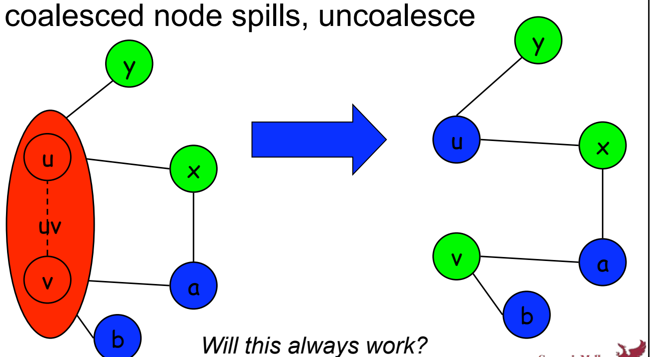
So:

- Use George if one of a & b has very large degree
- Use Briggs otherwise

Optimistic Coalescing

Aggressively coalesce

If coalesced node spills, uncoalesce



Will this always work?

Steps in register allocation

Which one is better?

Alternative Allocators

Graph allocator, as described, has issues

- What are they?

Alternative: Single pass graph coloring

- Build, Simplify, Coalesce as before
- In select, if can't color with register, color with stack location
  - Keep going
- Requires second, reload phase
  - "fixes" spilled variables
  - Might require that we reserve a register
  - Can get messy

Claim: Does a pretty good job

- Why?
  - Key is order nodes are colored (top-down)

Advantages? Disadvantages?

Alternative Allocators

Local/Global Allocation

gcc's approach, unless -fnew-ra

- Allocate "local" pseudo-registers
  - Lifetime contained within basic block
  - Register sufficiency no longer NP-Complete!
- Allocate global pseudo-registers
  - Single pass global coloring
  - Coloring heuristic may reverse local allocation
- Reload pass to fix spills (allocator does not generate spill code)
- Can also do global then local
- Advantages? Disadvantages?

Alternative Allocators

Linear Scan

- Performs single sweep over live ranges
- Each range represented by a single interval
- Greedily allocates/spills

Second Chance Binpacking

- maintains more state
  - lifetime holes, register-memory consistency
- will split a live range
- less greedy; may reevaluate previous allocations

Advantages? Disadvantages?

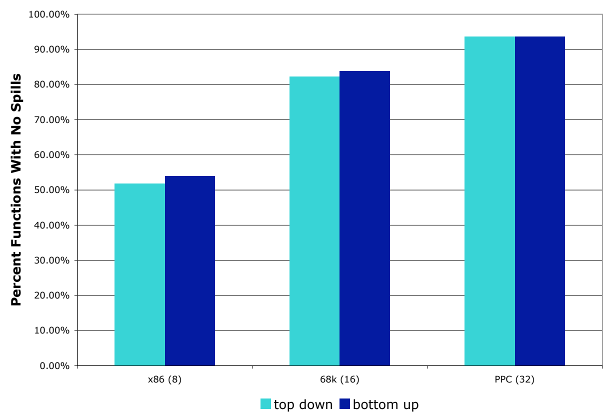
In Chaitin's words

"…since I was a mathematician, the register allocation kept getting simpler and faster as I understood better what was required. I preferred to base algorithms on a simple, clean idea that was intellectually understandable rather than write **complicated *ad hoc* computer code**…

So I regard the success of this approach, which has been the basis for much future work, as a triumph of the power of a simple mathematical idea over ad hoc hacking. Yes, **the real world is messy and complicated**, but one should try to base algorithms on clean, comprehensible mathematical ideas and only complicate them when absolutely necessary. In fact, certain instructions were omitted from the 801 architecture because they would have unduly complicated register allocation…"

— G. Chaitin, 2004

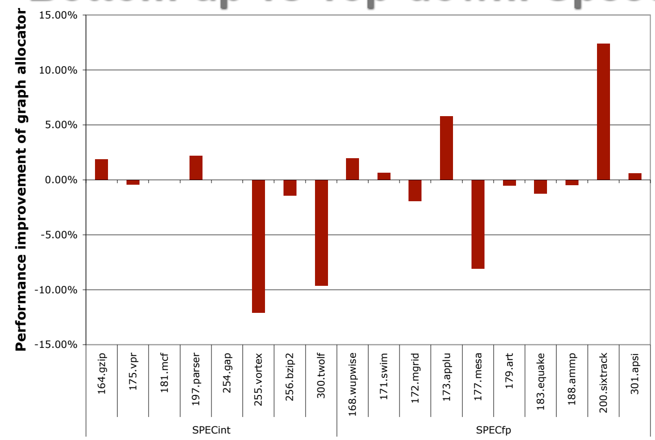
Avoiding Spills



1.8 Ghz Pentium 4; -O3 -funroll-loops *-fnew-ra*; gcc version 3.2.2

Carnegie Mellon
School of Computer Science

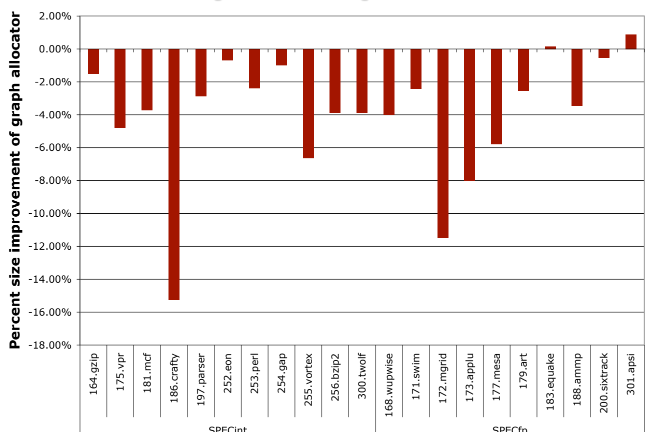
Bottom-up vs Top-down: Speed



1.8 Ghz Pentium 4; -O3 -funroll-loops; gcc version 3.2.2

Carnegie Mellon
School of Computer Science

Bottom-up vs Top-down: Size



x86; -Os; gcc version 3.2.2

Carnegie Mellon
School of Computer Science

Complexity of Register Allocation

Complexity of Register Allocation

Graph color is NP-complete

- what does this tell us about register allocation?

Given arbitrary graph can construct program with matching interference graph[1]

- simply determining if spilling is necessary is therefore NP-complete… or is it?

Can exploit structure of reducible program[2,3,4]

[1] G.J. Chaitin, M. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P. Markstein. Register allocation via coloring. Computer Languages, 6:47-57, 1981.
[2] H. Bodlaender, J. Gustedt, and J. A. Telle, "Linear-time register allocation for a fixed number of registers," in *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, pp. 574–583, Society for Industrial and Applied Mathematics, 1998.
[3] S. Kannan and T. Proebsting, "Register allocation in structured programs," in *Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pp. 360–368, Society for Industrial and Applied Mathematics, 1995.
[4] M. Thorup, "All structured programs have small tree width and good register allocation," *Inf. Comput.*, vol. 142, no. 2, pp. 159–181, 1998.

Complexity of Register Allocation

Complexity of local register allocation?

- linear algorithm for register sufficiency

SSA Form?

- interference graph is turns out to be both perfect[1] and chordal[2]
  - can color in linear time
- BUT all bets are off after SSA elimination[3]

[1] Philip Brisk, Foad Dabiri, Jamie Macbeth, and Majid Sarrafzadeh. Polynomial time graph coloring register allocation. In 14th International Workshop on Logic and Synthesis. ACM Press, 2005.
[2] Sebastian Hack. Interference graphs of programs in SSA-form. Technical Report ISSN 1432-7864, Universitat Karlsruhe, 2005.
[3] Jens Palsberg and Fernando Magno Quintao Pereira Register allocation after classical SSA elimination is NP-complete, In Proceedings of FOSSACS'06, Foundations of Software Science and Computation Structures. Springer-Verlag (LNCS), Vienna, Austria, March 2006.

Complexity of Register Allocation

Complexity of optimizing spill code?

- NP-complete even without control flow[1]

Complexity of optimal coalescing?

- NP-complete[2]

[1] Martin Farach and Vincenzo Liberatore. On local register allocation. In 9th ACMSIAM symposium on Discrete Algorithms, pages 564 { 573. ACM Press, 1998.
[2] Andrew W. Appel and Lal George. Optimal spilling for cisc machines with few registers. In Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation, pages 243–253. ACM Press, 2001.