Register Allocation II

15-745 Optimizing Compilers

Spring 2006

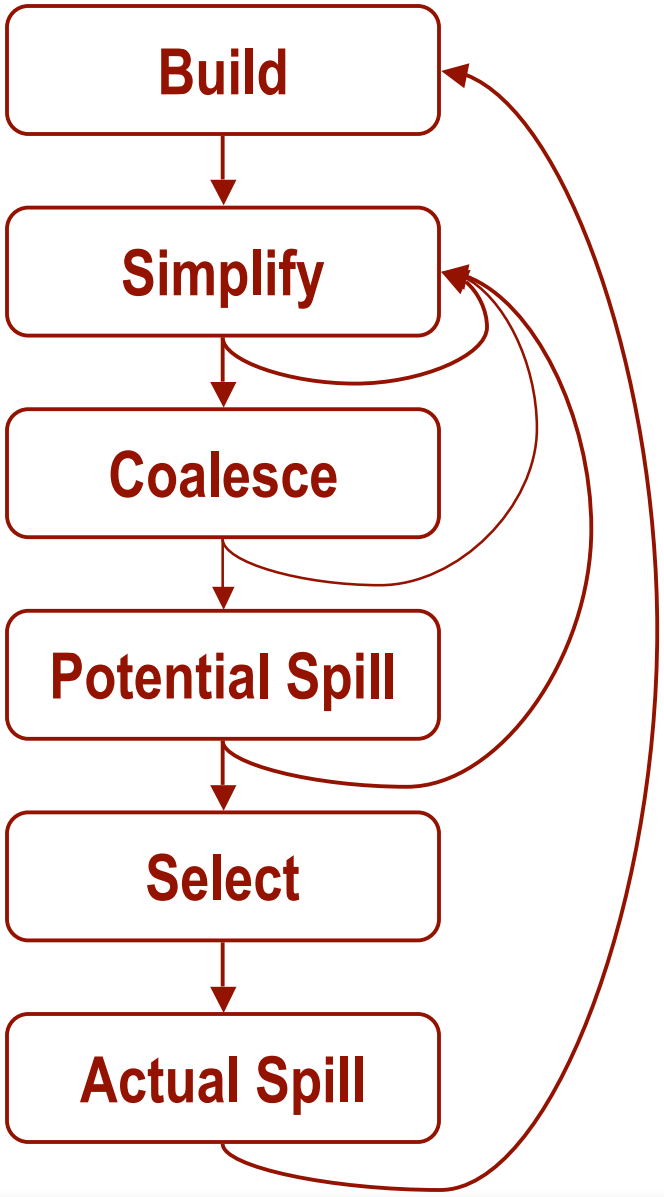
David Koes

Carnegie Mellon
School of Computer Science

Outline

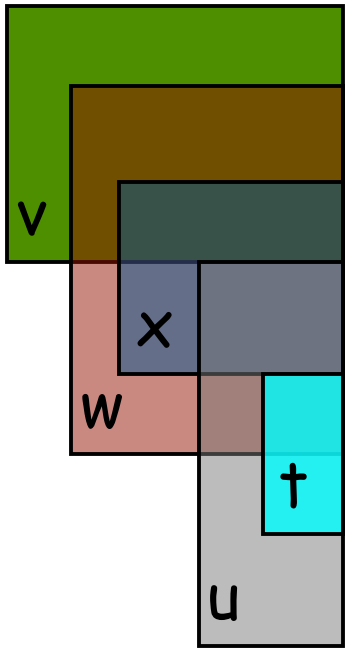
- Review
- Old Subtleties
- New Subtleties

Review


Build
Simplify
Coalesce
Potential Spill
Select
Actual Spill

Build

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



First compute live ranges

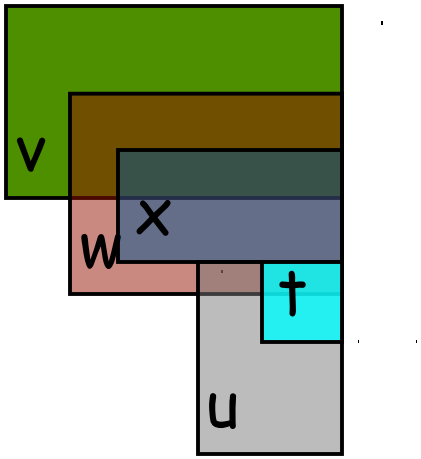
- use **both** reaching definitions and liveness information
- live range defined by definition point
- ends when variable dies

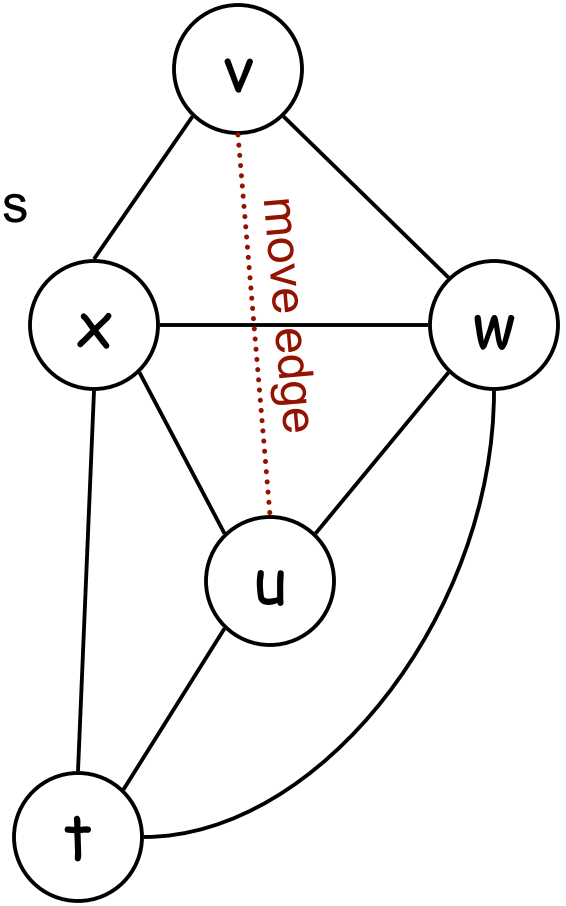
Build

Construct interference graph:

- each node represents a live range
- edges represent live ranges that overlap
- put in move edges between move operands

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

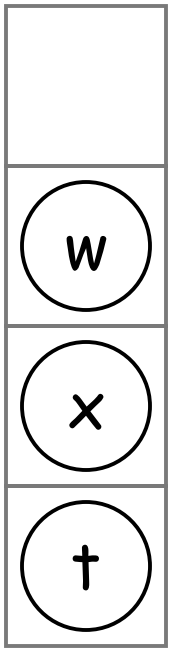




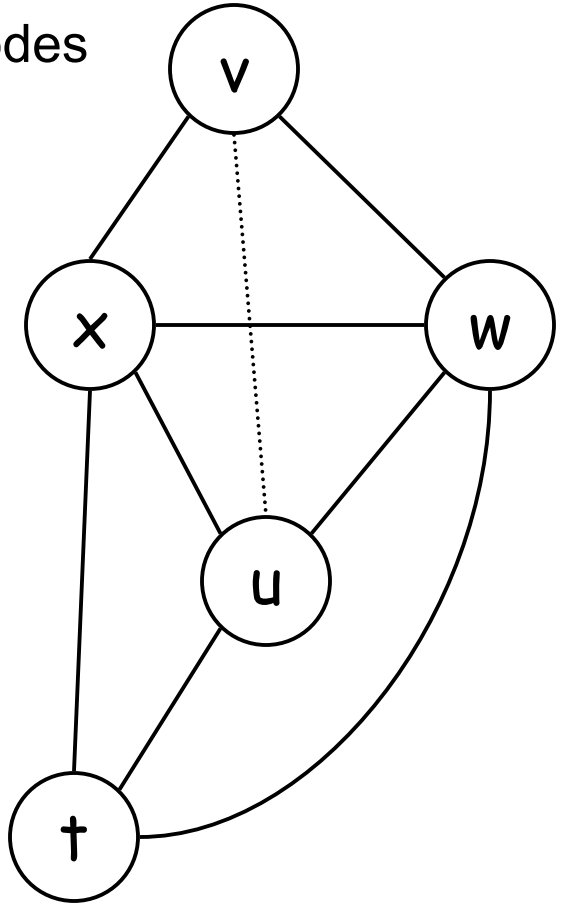
Simplify

Reduce the graph:

- remove non-move related, easy to color, nodes
- easy to color: degree < k
- place on stack



k = 4



Coalesce

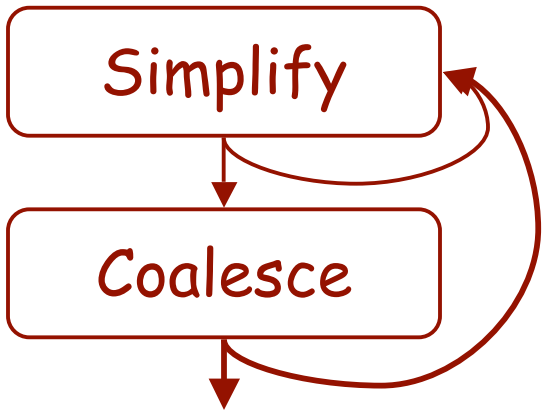
Coalesce moves:

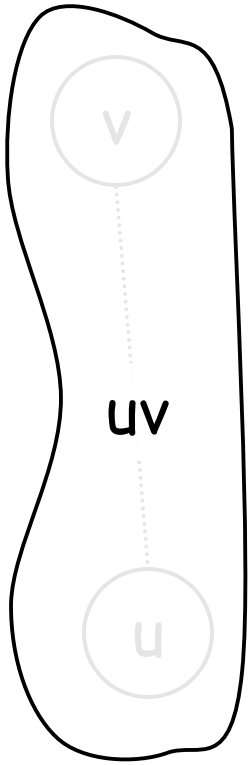
- conservatively combine operands of a move

- subtlety: how?

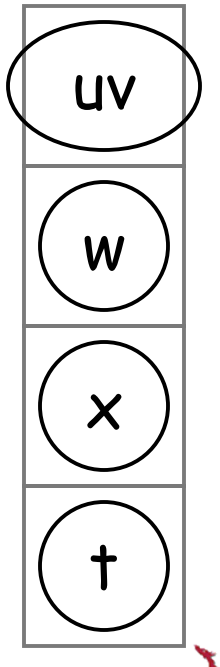
Repeat Simplify

-Detail: If both Simplify and Coalesce get stuck, start simplifying move related nodes





k = 4



What if we can't simplify?

Now what?

Be optimistic:

- Put a node with degree ≥ k on stack

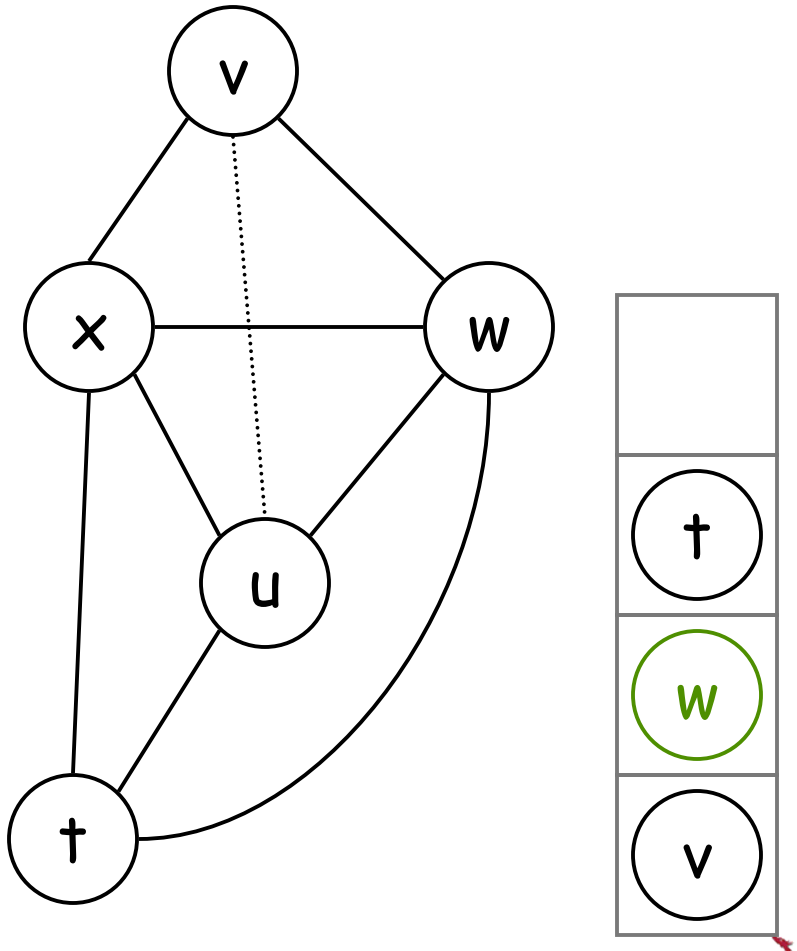
- Lose guarantee that anything we put on stack is colorable

- 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



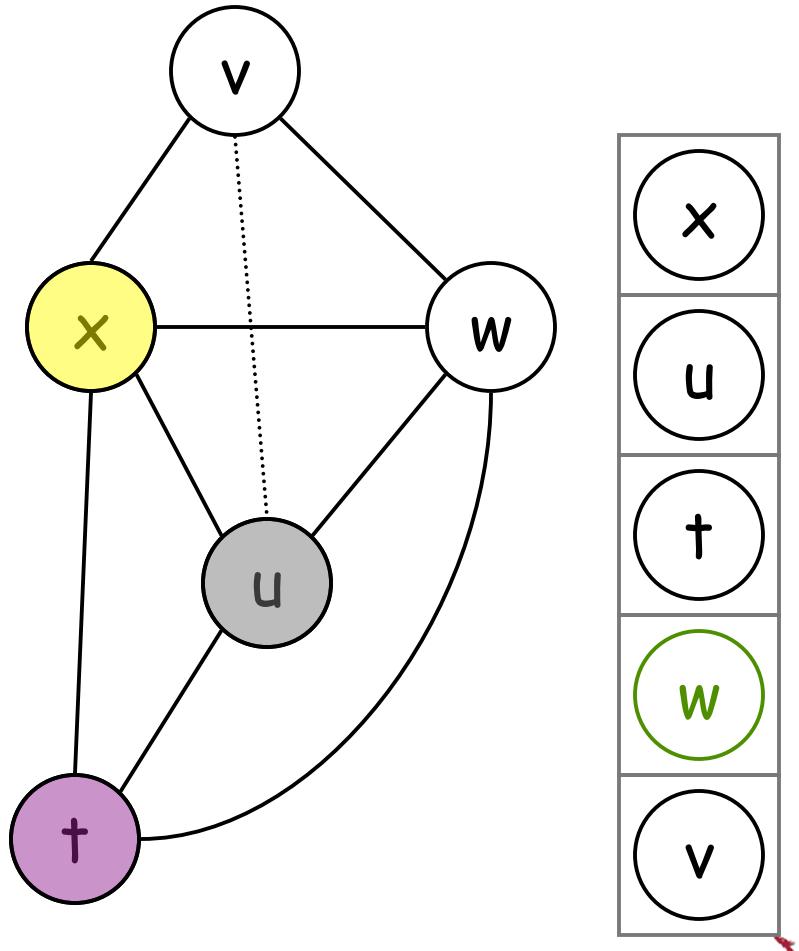
Select

Pop a node from the stack

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

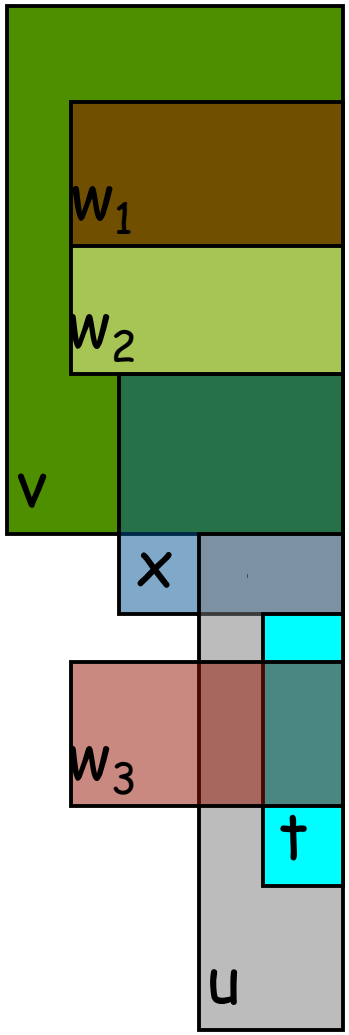
This will always be possible, *unless* the node is a potential spill

If it is not possible spill variable and rebuild graph



Spilling

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

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

Rematerialization

An alternative to spilling

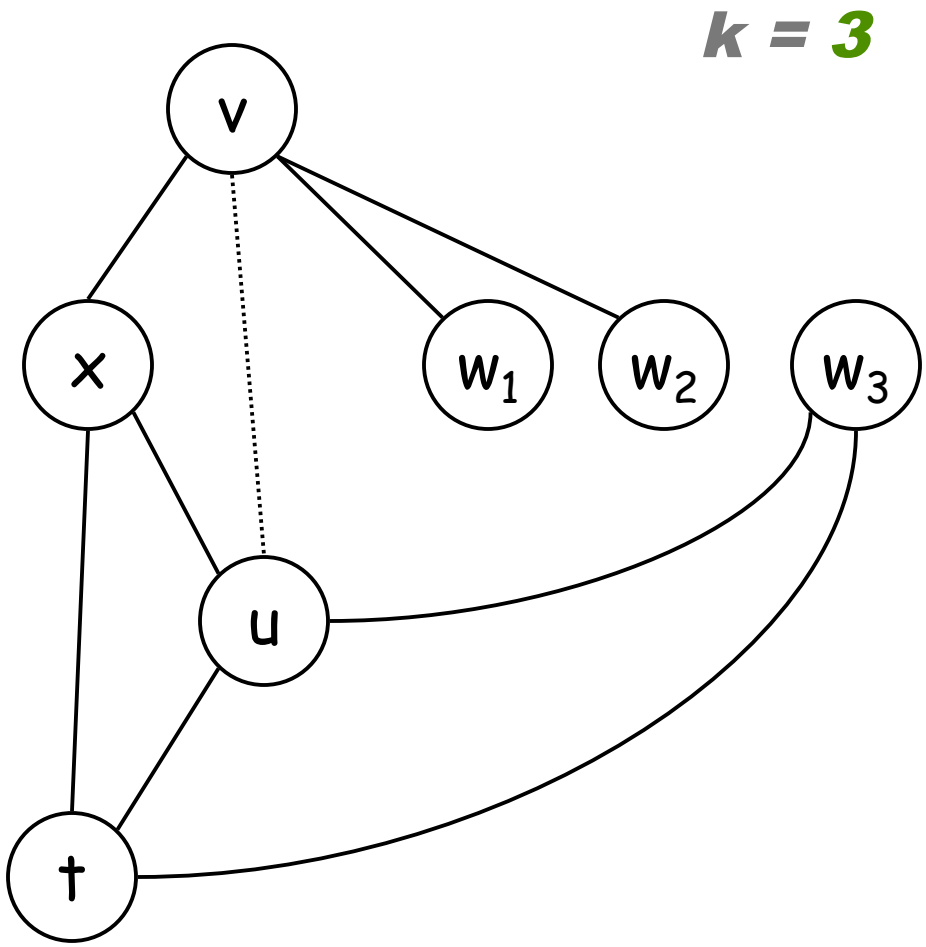
- Recompute value of variable instead of store/load to memory
- Example:

```
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
```

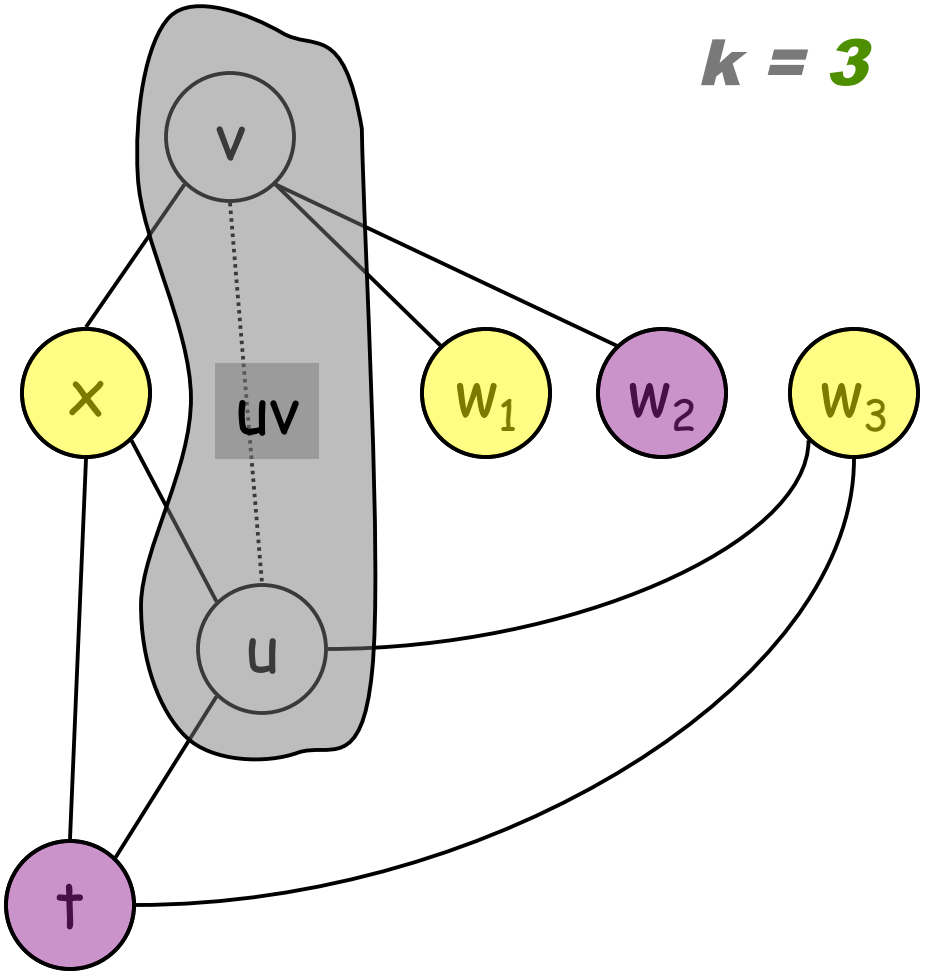
Build Take Two

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

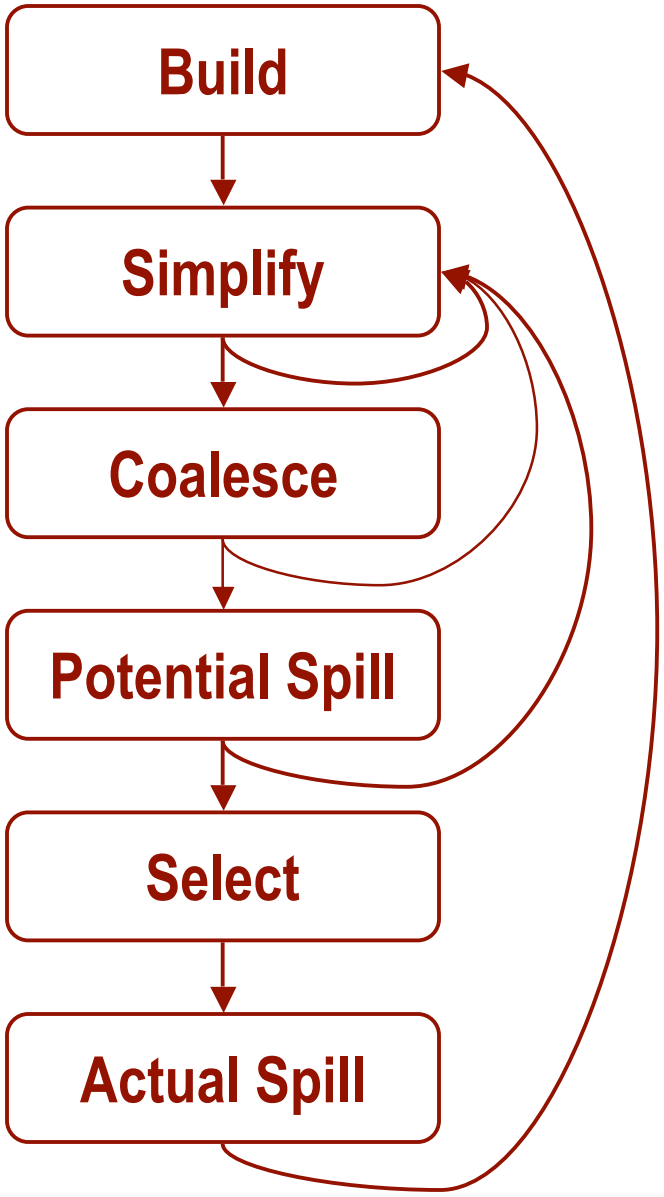


Recalculate interference graph

Simplify->Coalesce->Select



Review


Build
Simplify
Coalesce
Potential Spill
Select
Actual Spill

Old Subtleties

1. MOVE instructions
2. Merging live ranges
3. Splitting live ranges
4. Choosing potential spills
5. Allocating spill slots
6. Coalescing is bad

MOVE Instructions

- During liveness analysis, MOVE instructions should be treated specially

```
t := s
 …
x <- ⊗(…s…)
 …
y <- ⊗(…t…)
```

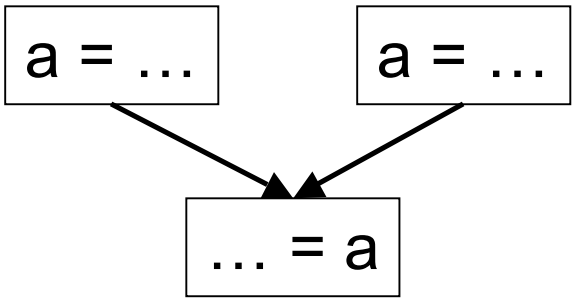
Note that s and t don't really interfere

Under what conditions can we remove the interference edge between s *and* t*?*

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.

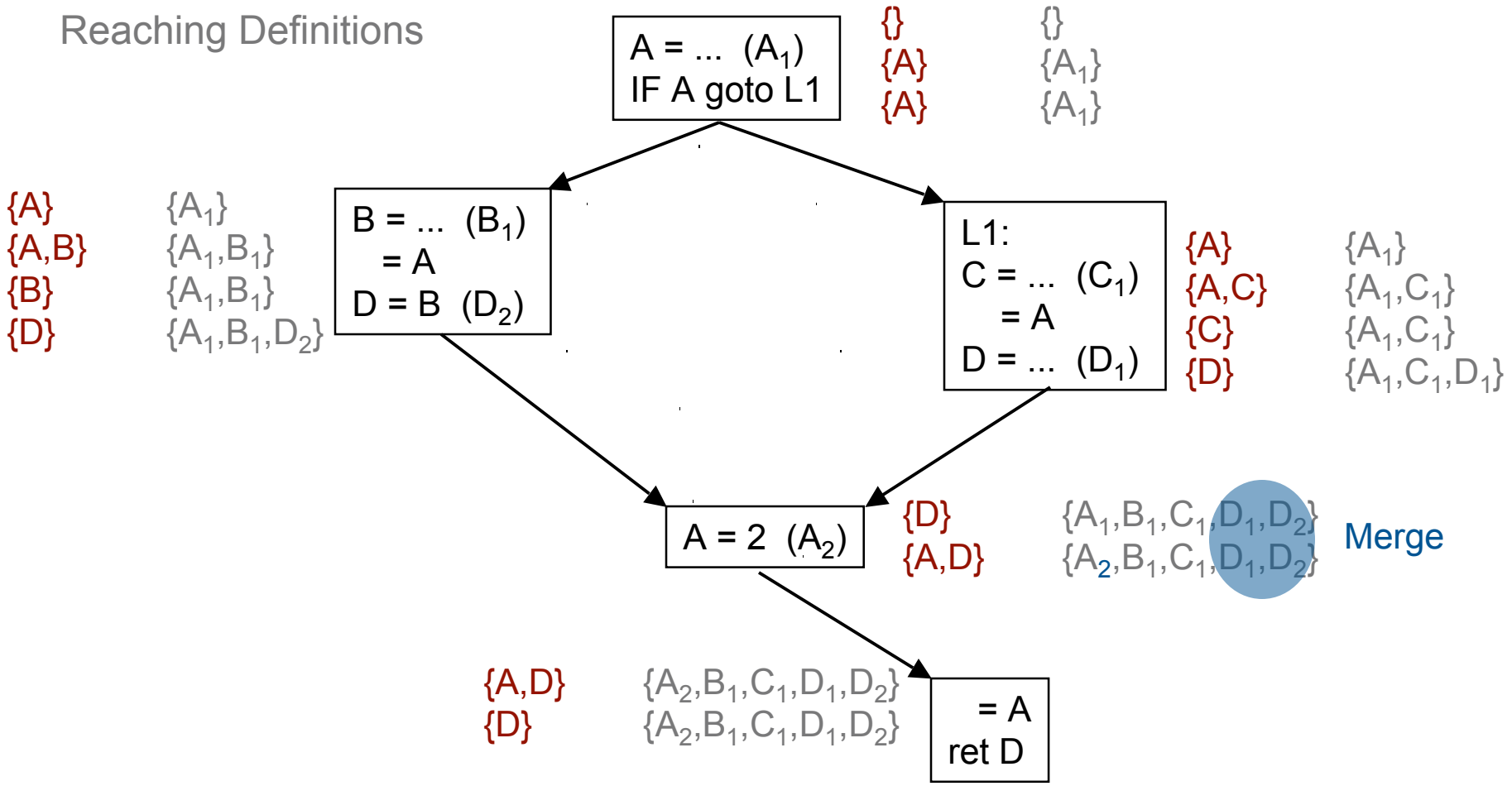
– How to compute a live range?



Two overlapping live ranges for the same variable must be merged

Example

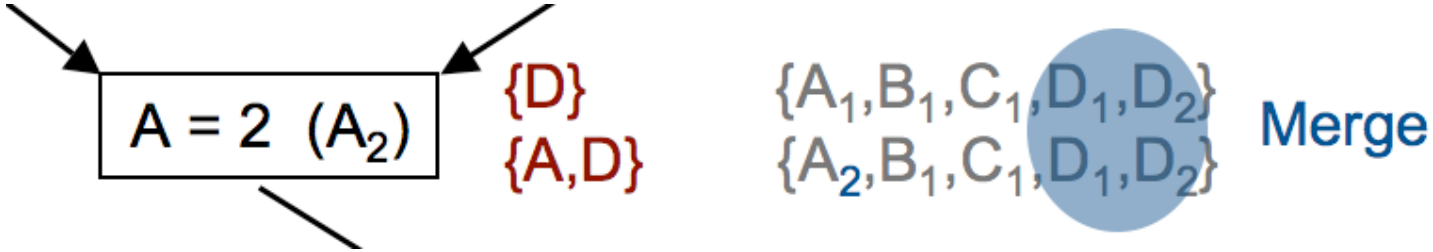
Live Variables
Reaching Definitions



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

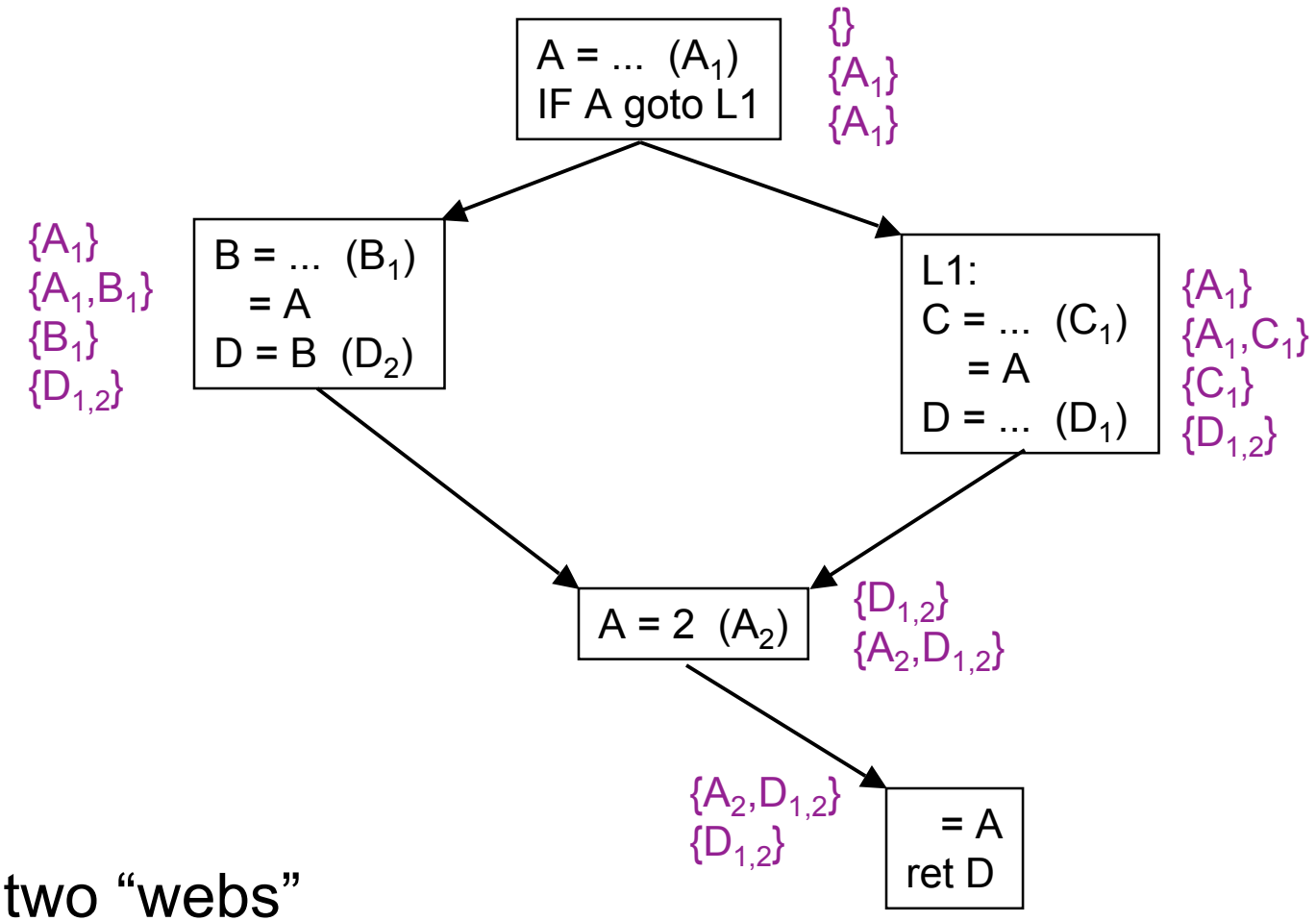
Merging Live Ranges

- 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"

Example: Merged Live Ranges



A has two "webs"

makes register allocation easier

Edges of Interference Graph

Intuitively:

Two live ranges (necessarily of different variables) may interfere if they overlap at some point in the program.

Algorithm:

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

An optimized definition & algorithm for edges:

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

Faster?

Better quality?

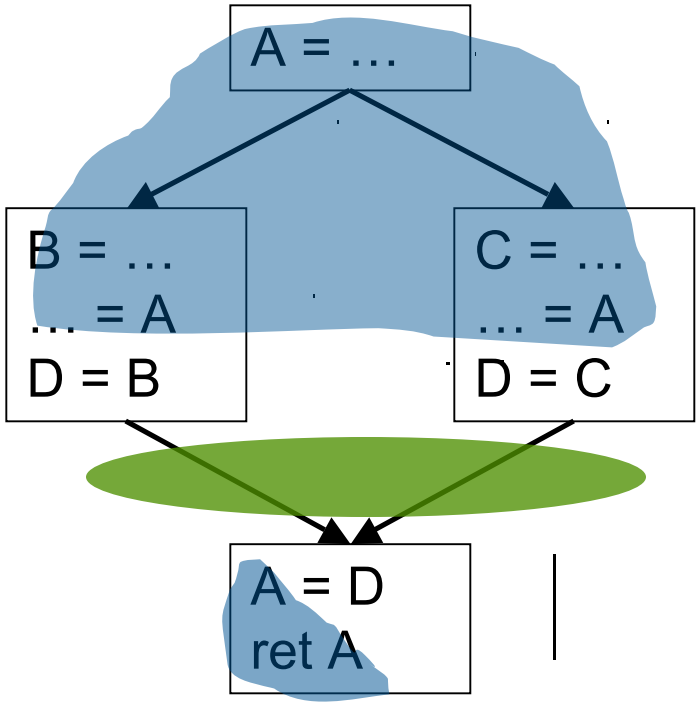
A = 2 (A_2) $\{D_{1,2}\}$ $\{A_2, D_{1,2}\}$ → Edge between A_2 and $D_{1,2}$

Reducing Register Pressure

Splitting variables into live ranges creates an interference graph that is easier to color

Eliminate interference in a variable's "dead" zones.

Increase flexibility in allocation:
can allocate same variable to different registers



Live Range Splitting

Split a live range into smaller regions (by paying a small cost) to create an interference graph that is easier to color

Eliminate interference in a variable's nearly dead zones.

Cost: Memory loads and stores
Load and store at boundaries of regions with no activity

active live ranges at a program point **can be** > # registers

Can allocate same variable to different registers

Cost: Register operations a register copy between regions of different assignments

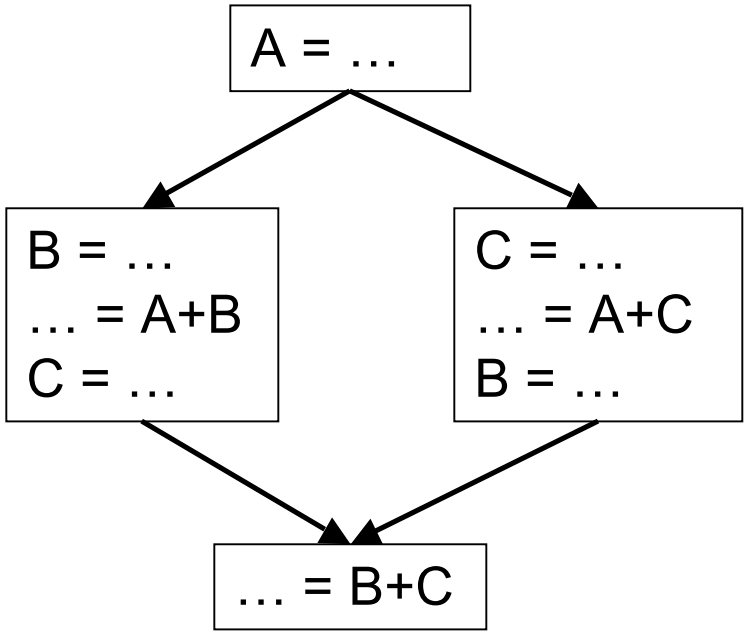
active live ranges **cannot be** > # registers

Examples

Example 1:

```
A = …; B = …;
FOR i = 0 TO 10
   FOR j = 0 TO 10000
        A = A + ...
        (does not use B)
    FOR j = 0 TO 10000
        B = B + ...
        (does not use A)
```

Example 2:



One Algorithm

- **Observation: Spilling is absolutely necessary if**
  __not__ degree in graph
  - number of live ranges active at a program point > n

- **Apply live-range splitting before coloring**
  - Identify a point where number of live ranges > n
  - For each live range active around that point
    - find the outermost “block construct” that does not access the variable
  - Choose a live range with the largest inactive region
  - Split the inactive region from the live range

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

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…

Spill coalescing

- On machines with few registers (like the Pentium), a lot of temps can get spilled
  - activation records get big
  - memory-to-memory moves get generated
- For these reasons, it is good to do *spill coalescing*

Spill coalescing

- Spill coalescing should be done right after the **Select** phase (ie, before generating spill code)
  - for all instructions of the form
    - MOVE(t1,t2)
  - where t1 and t2 are spilled:
    - if t1 and t2 don't interfere, then coalesce
    - then, perform Simplify and Select to color all spilled nodes
      - the colors are stack slots

Move Coalescing

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

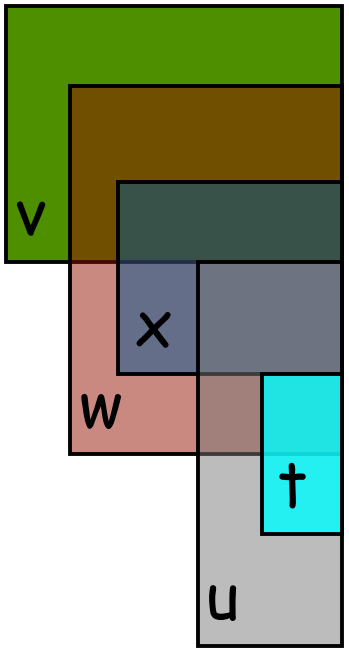
```
movl t1,t2          movl %eax,%eax          addl %edx,%eax
addl t3,t2    ➡     addl %edx,%eax    ➡
```

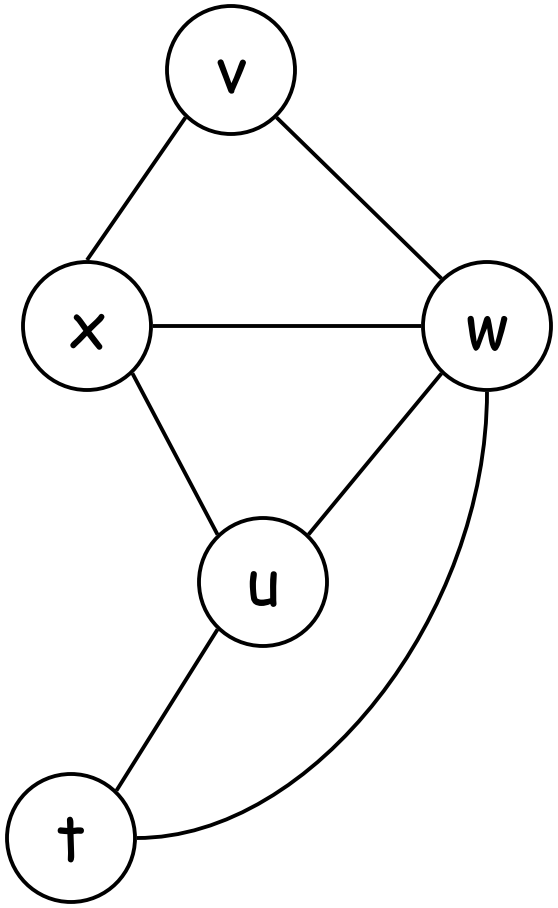
When can we coalesce t1 and t2?

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...

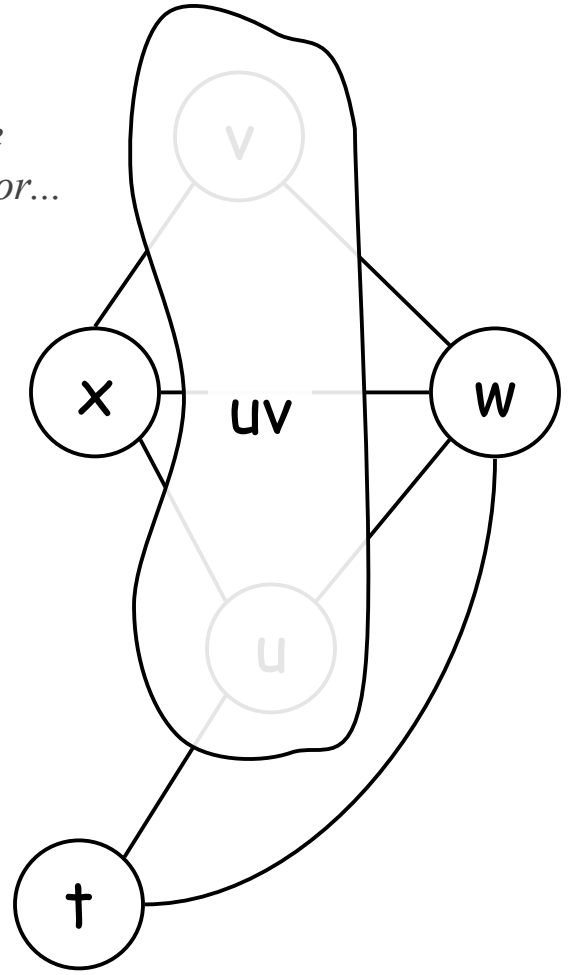
...then construct interference graph

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...

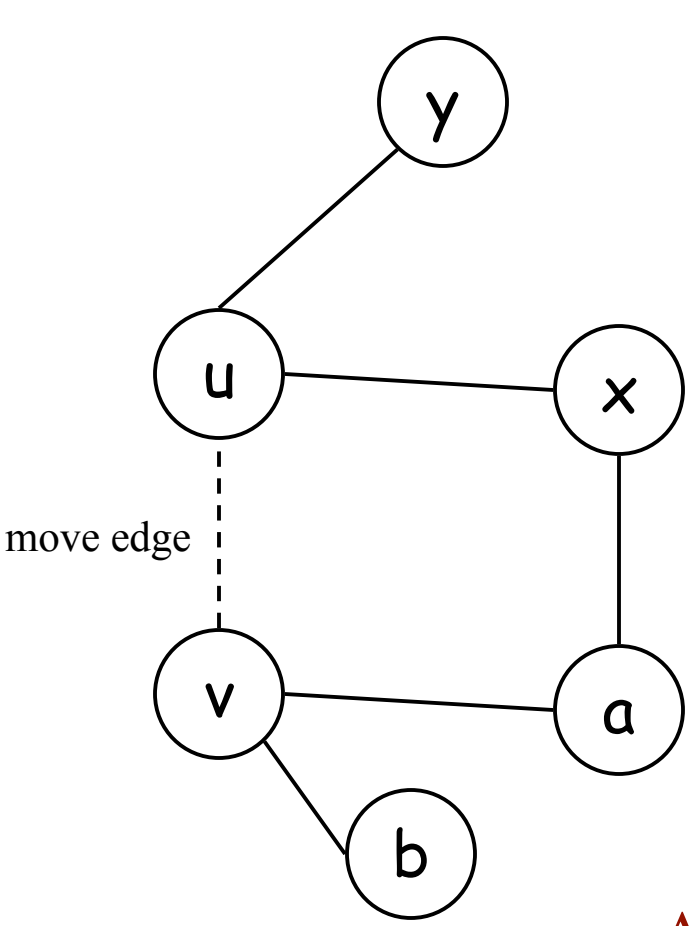
...merge u and v to form a single node



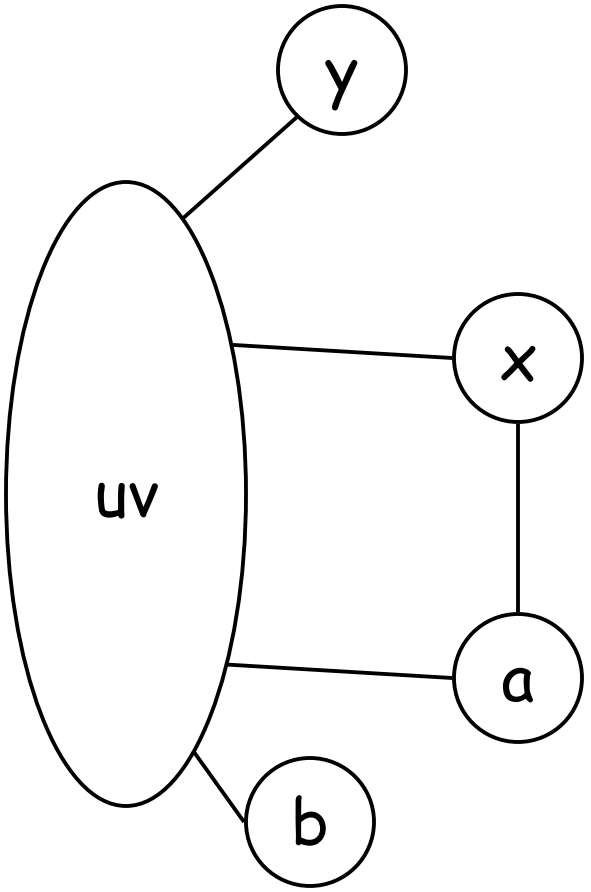
u and v are special:
A move whose source is not live-out of the move is a candidate for coalescing

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

Is Coalescing Always Good?



vs.



And the winner is?

2 colorable

3 colorable

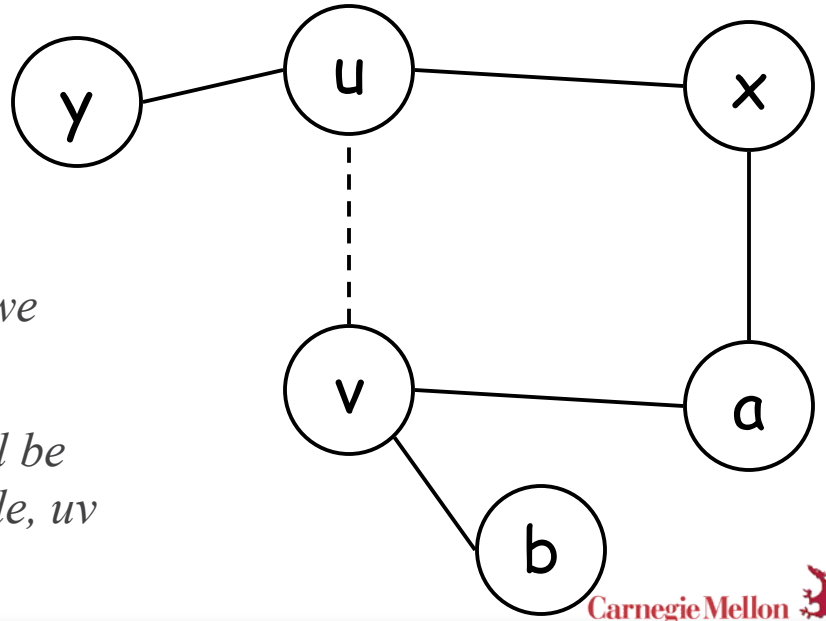
When should we coalesce?

Always

- If we run into trouble start un-coalescing
  - no nodes with degree < k, see if breaking up coalesced nodes fixes
- yuck

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

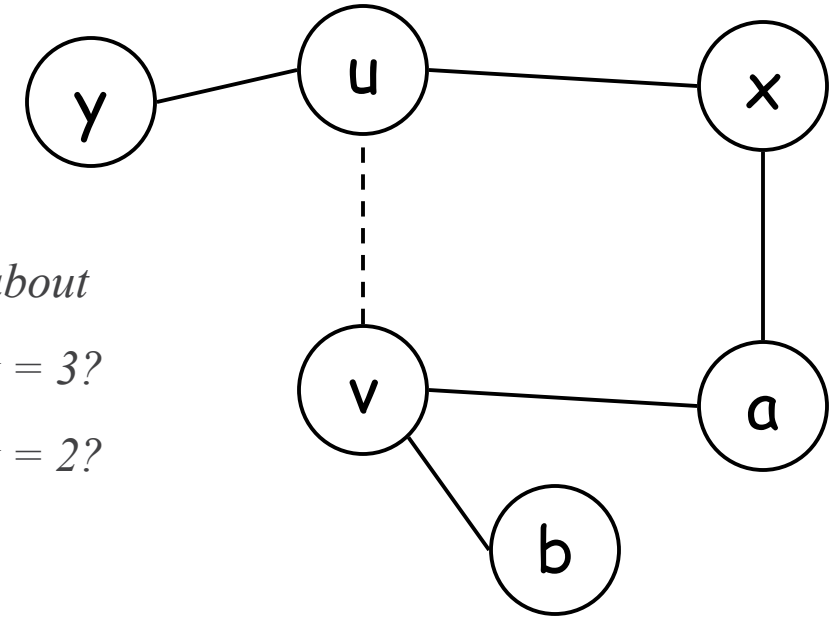
Why?

- –*Simplify* pass removes all nodes with degree < k
- –number 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,
  - degree of t < k

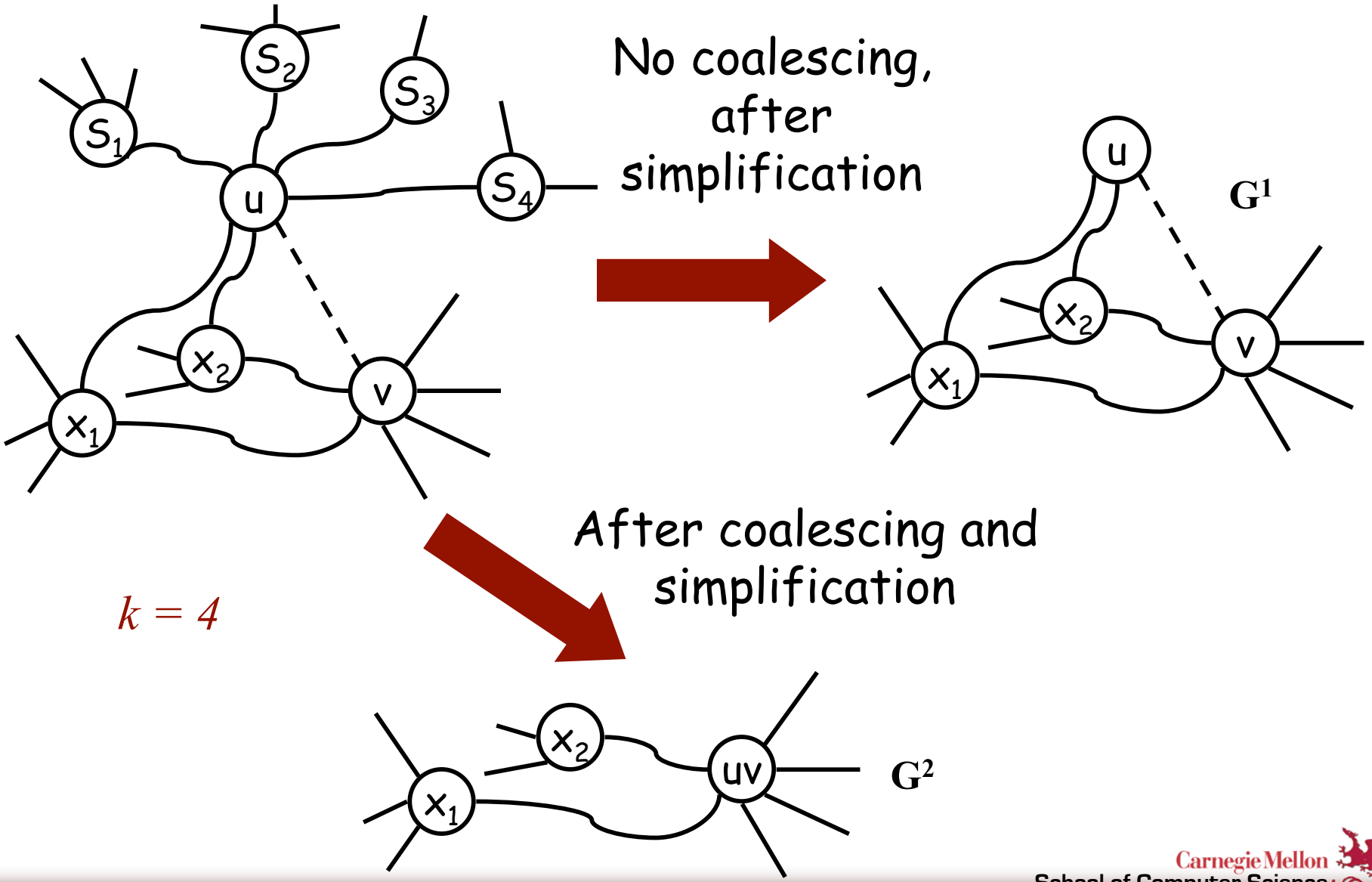
doesn't change degree

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

New Subtleties

- Alternative Algorithms
- Complexity of register allocation
- Effectiveness of graph coloring

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…

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 (Morgan)
- Advantages? Disadvantages?

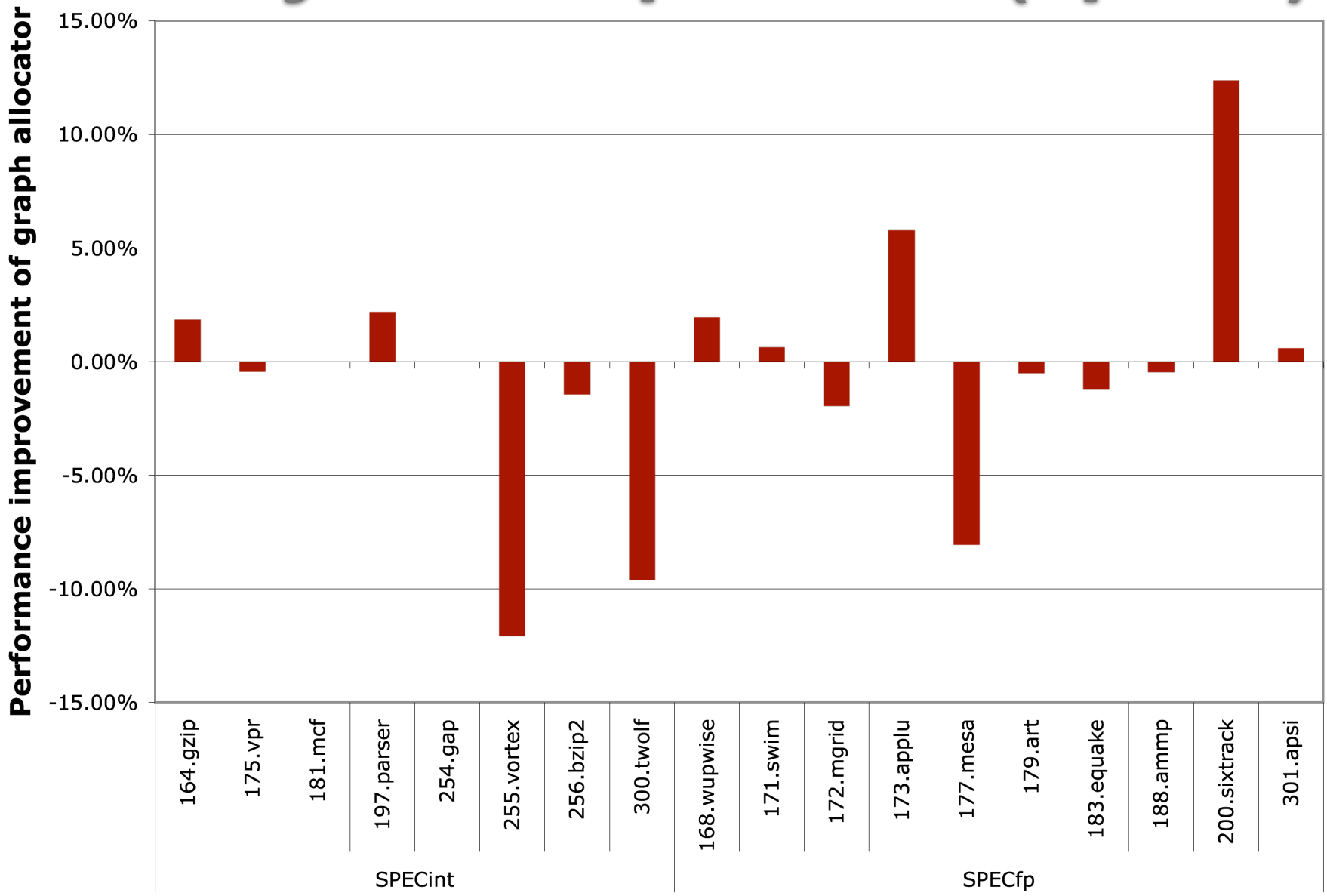
In Chaitin's words

"…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…"

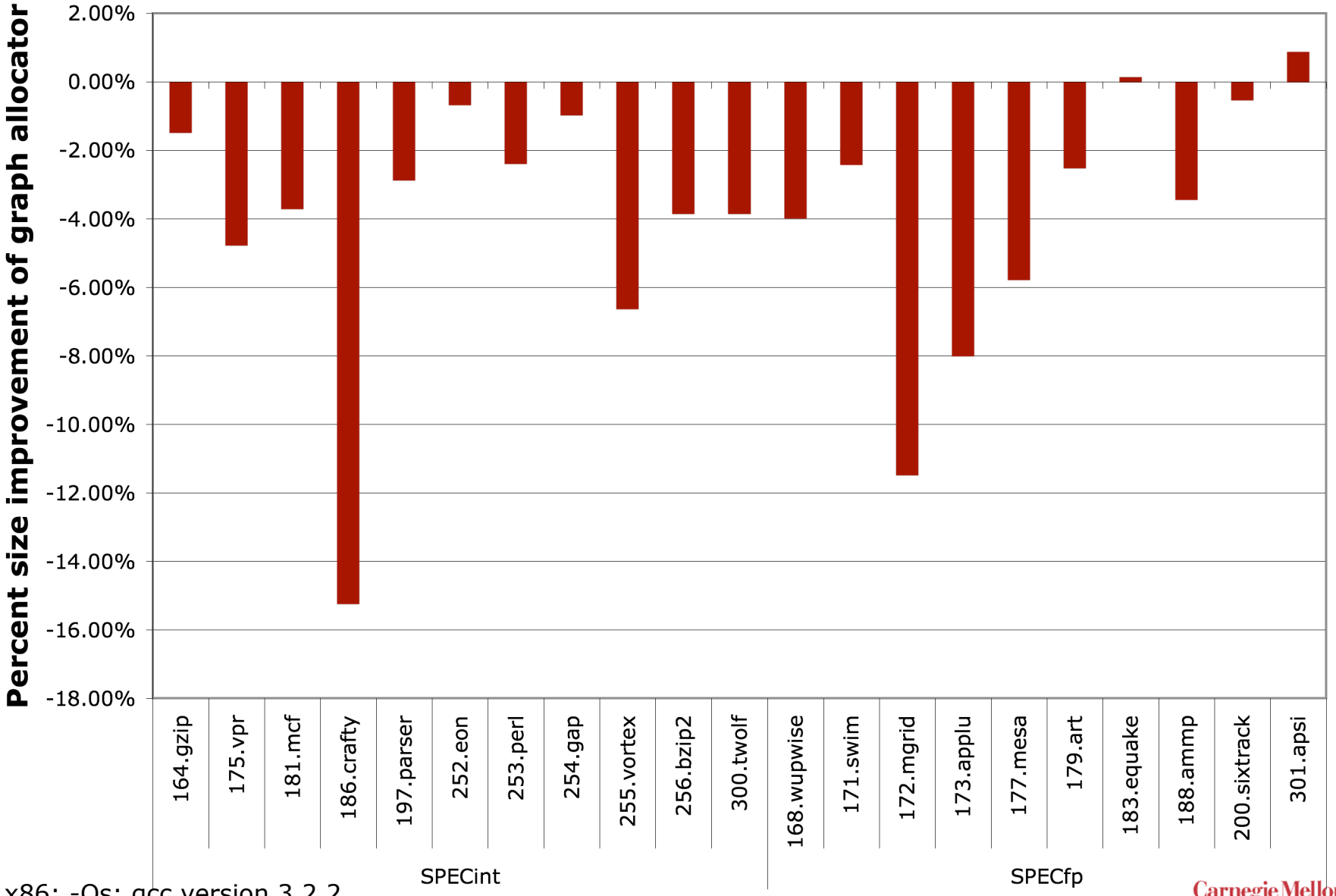
— G. Chaitin, 2004

Theory meets practice (speed)



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

Theory meets practice (size)



x86; -Os; gcc version 3.2.2

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.

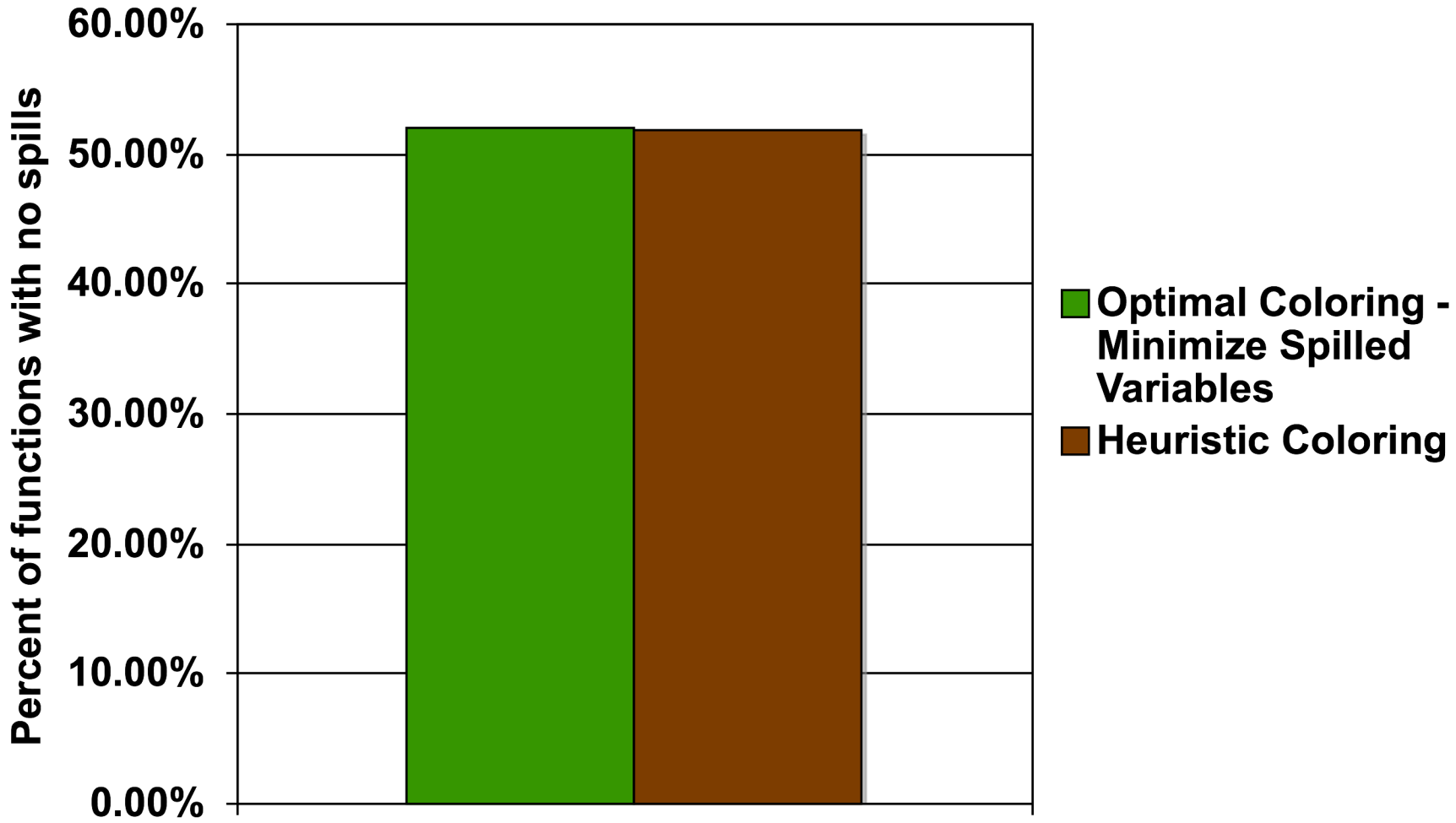
Complexity of Register Allocation

- 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.

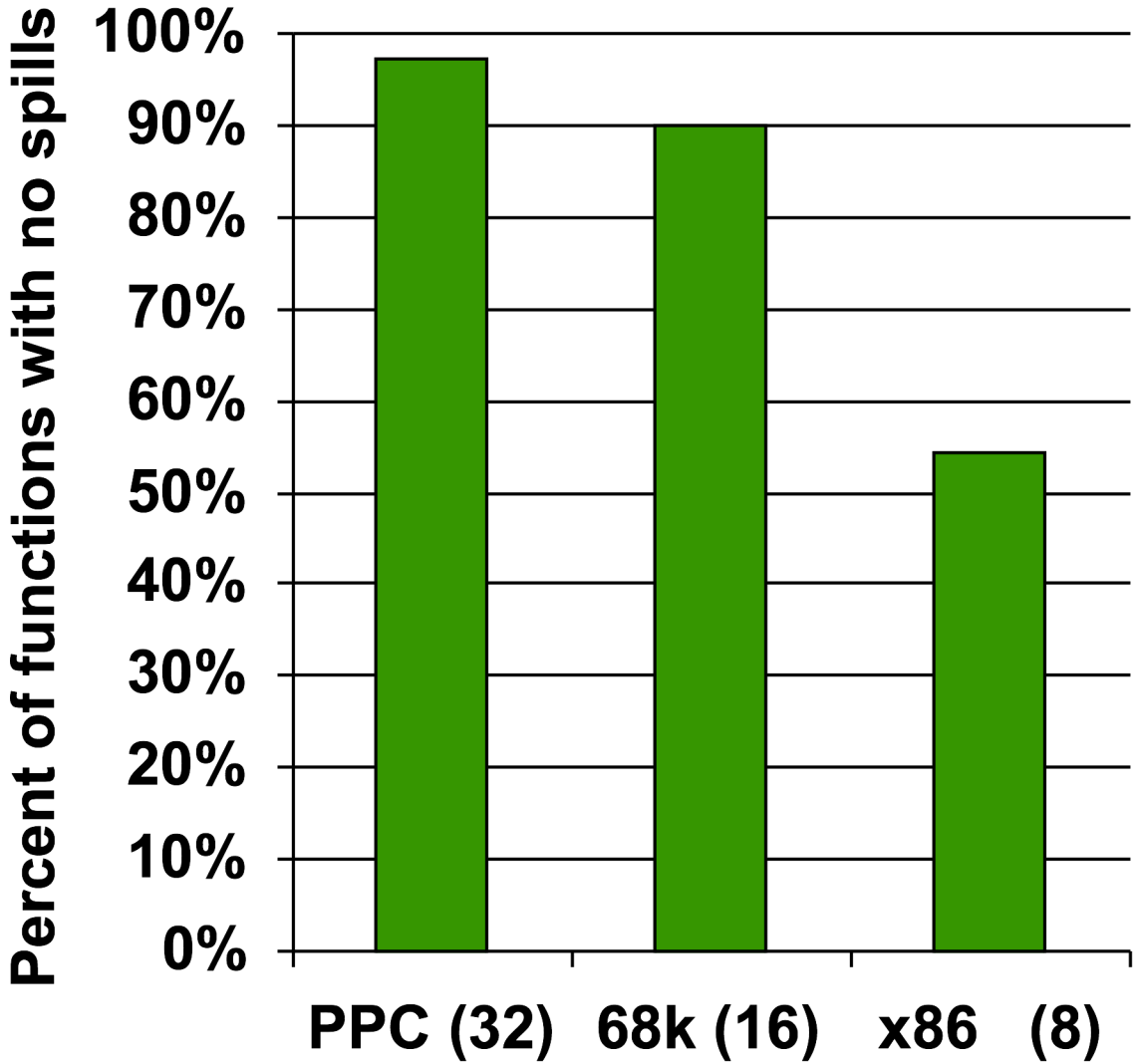
Effectiveness of Graph Coloring

Avoiding Spills



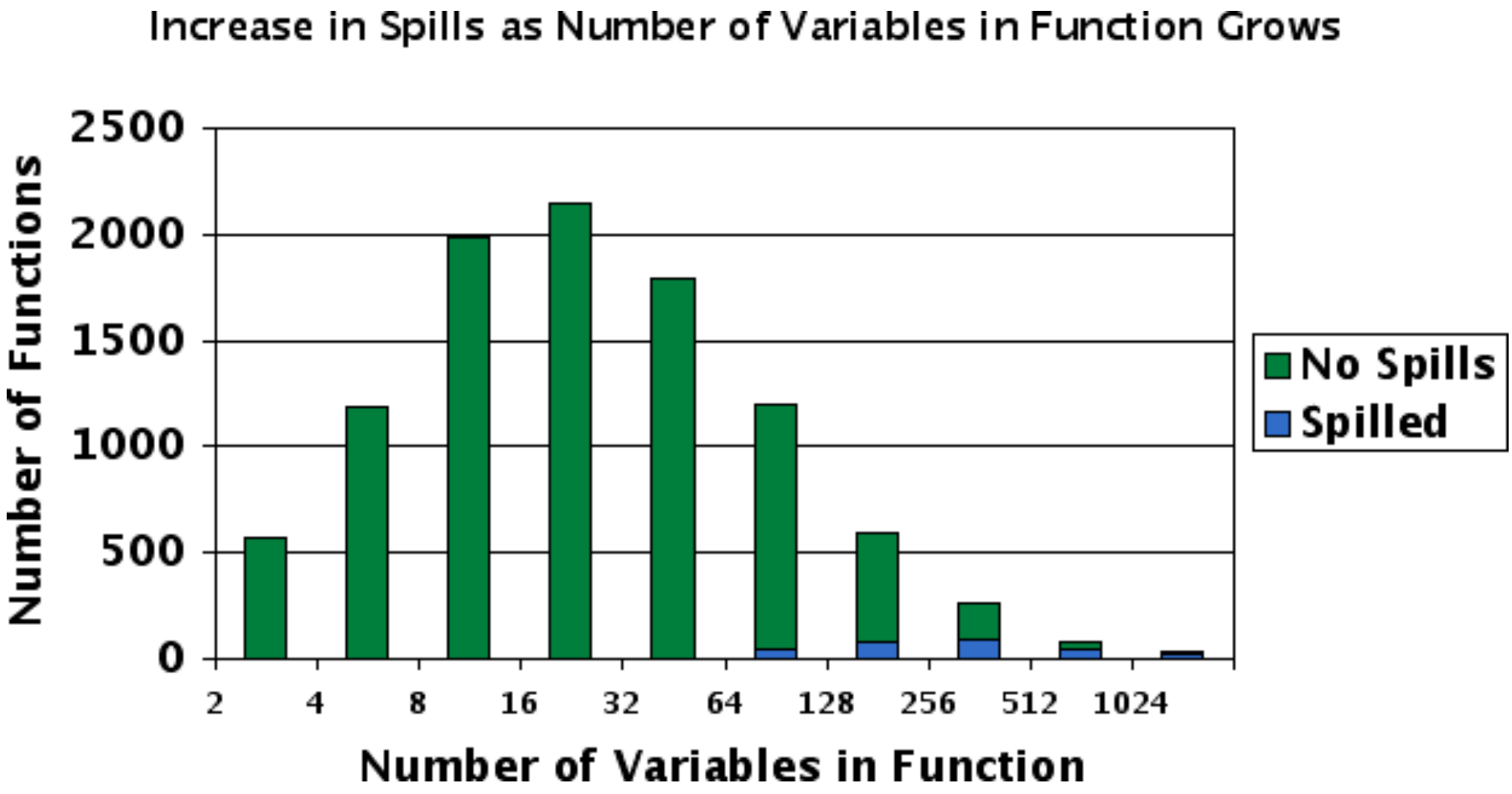
1.8 Ghz Pentium 4; -O3 -funroll-loops -fnew-ra; gcc version 3.2.2

Other architectures

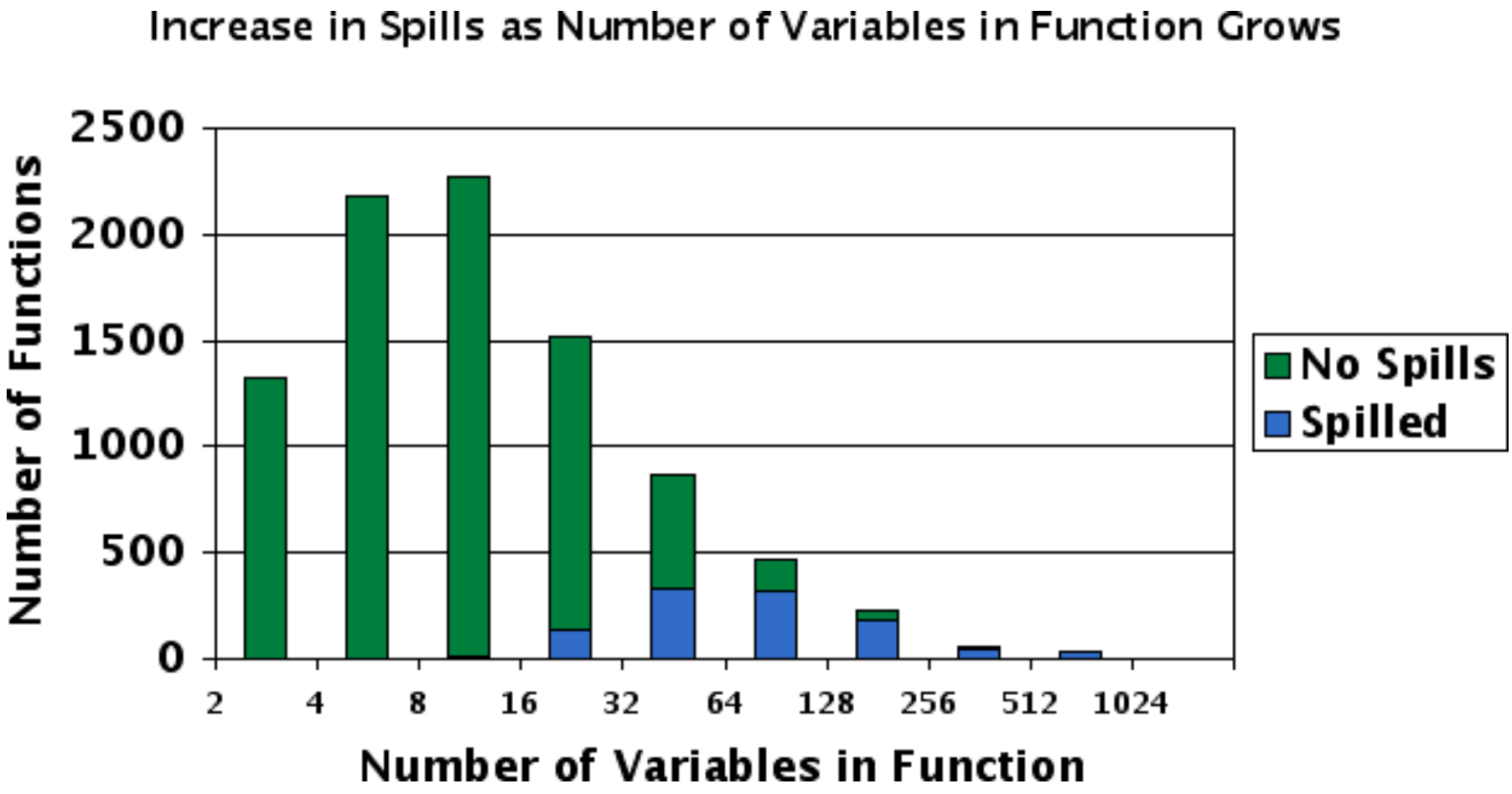


1.8 Ghz Pentium 4; -O3 -funroll-loops -fnew-ra; gcc version 3.2.2

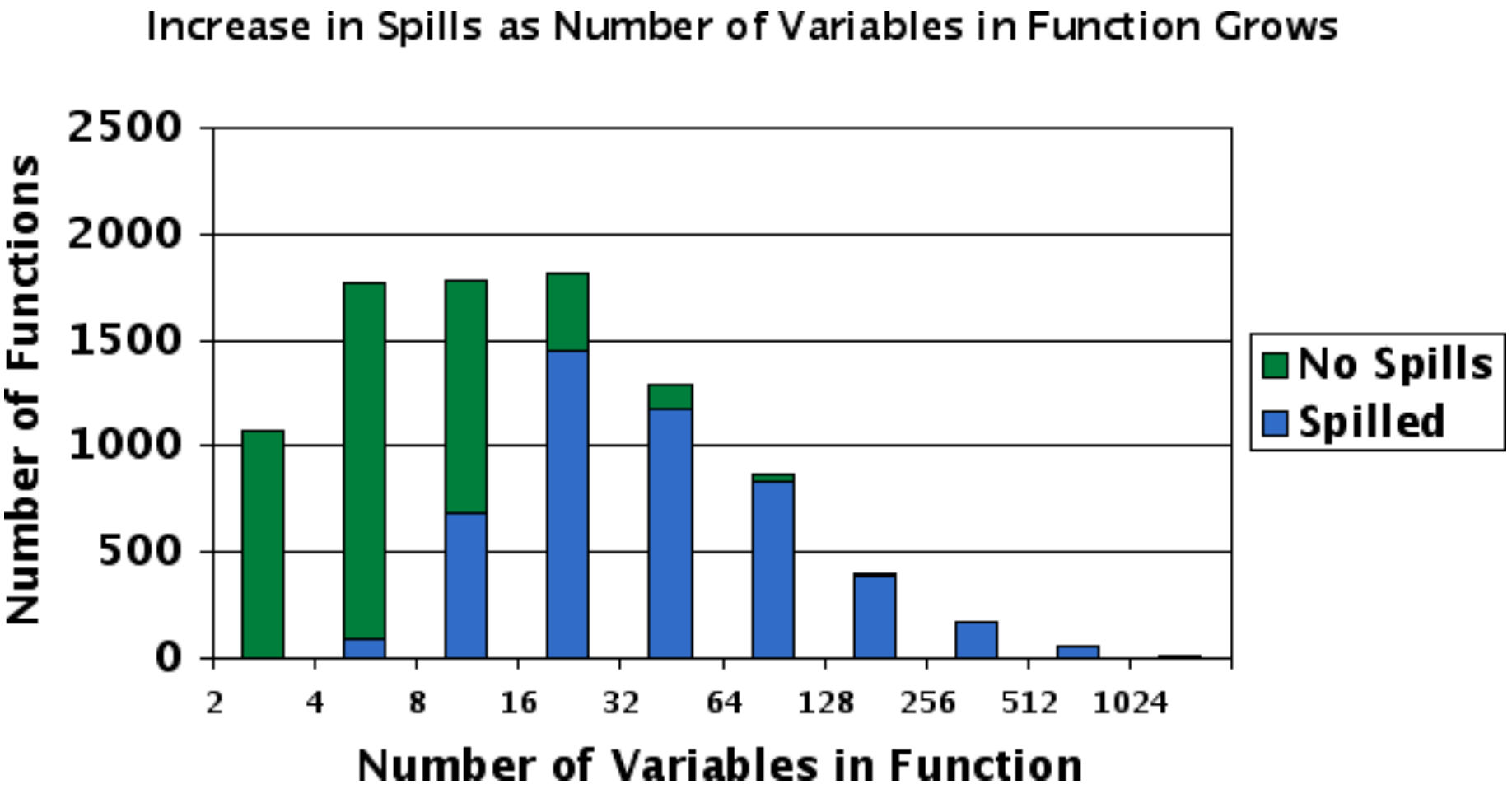
PPC (32 registers)



68k (16 registers)

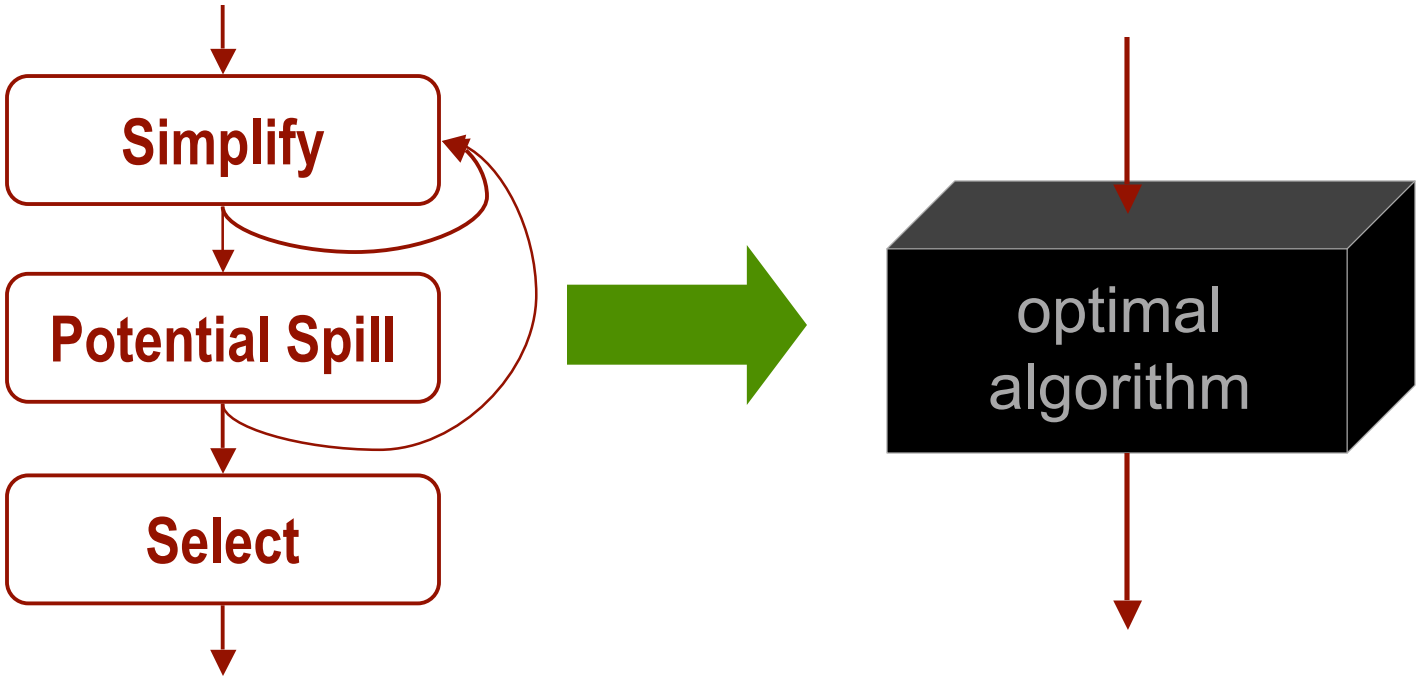


x86 (8 registers)

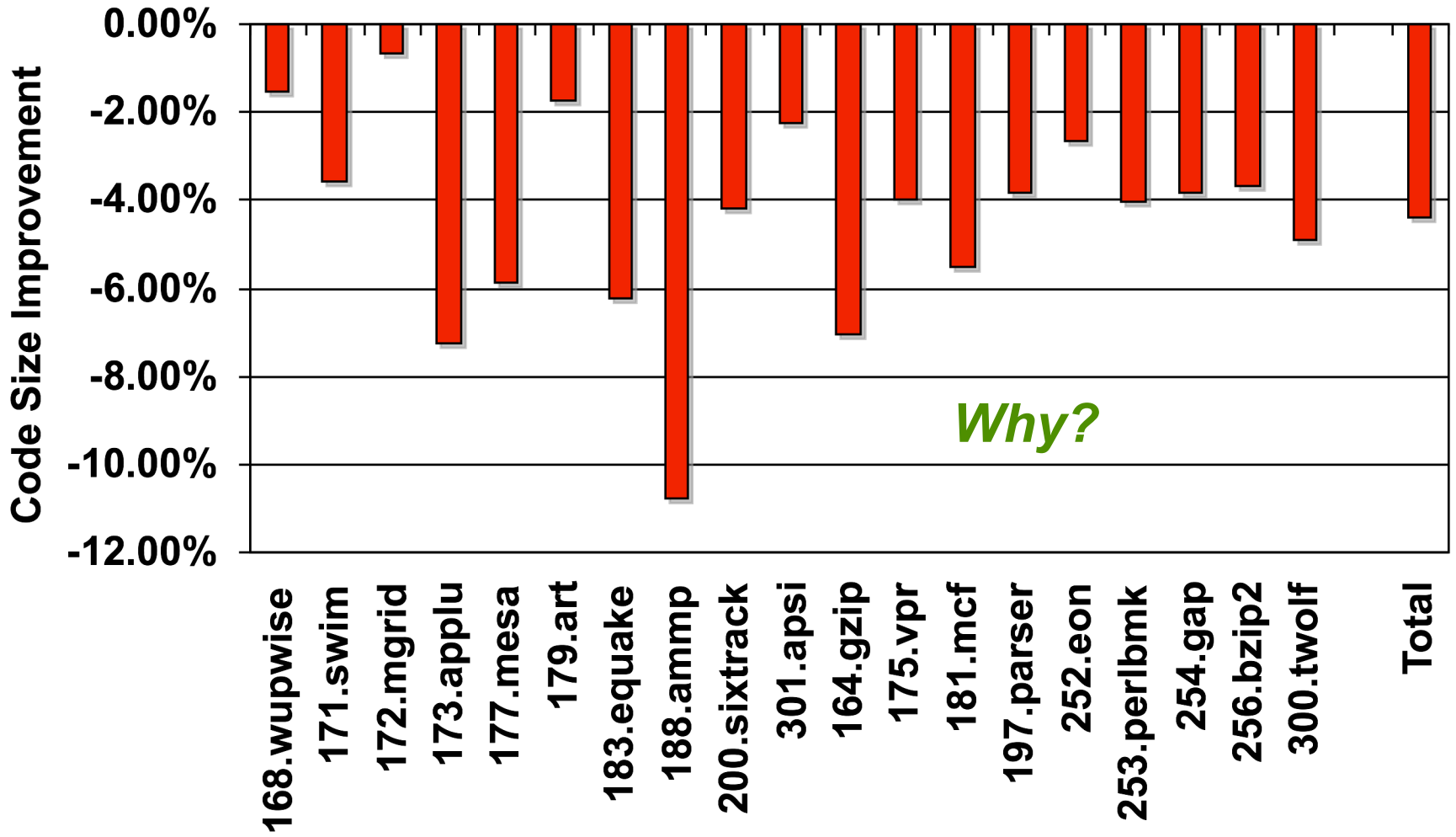


Importance of Coloring Quality

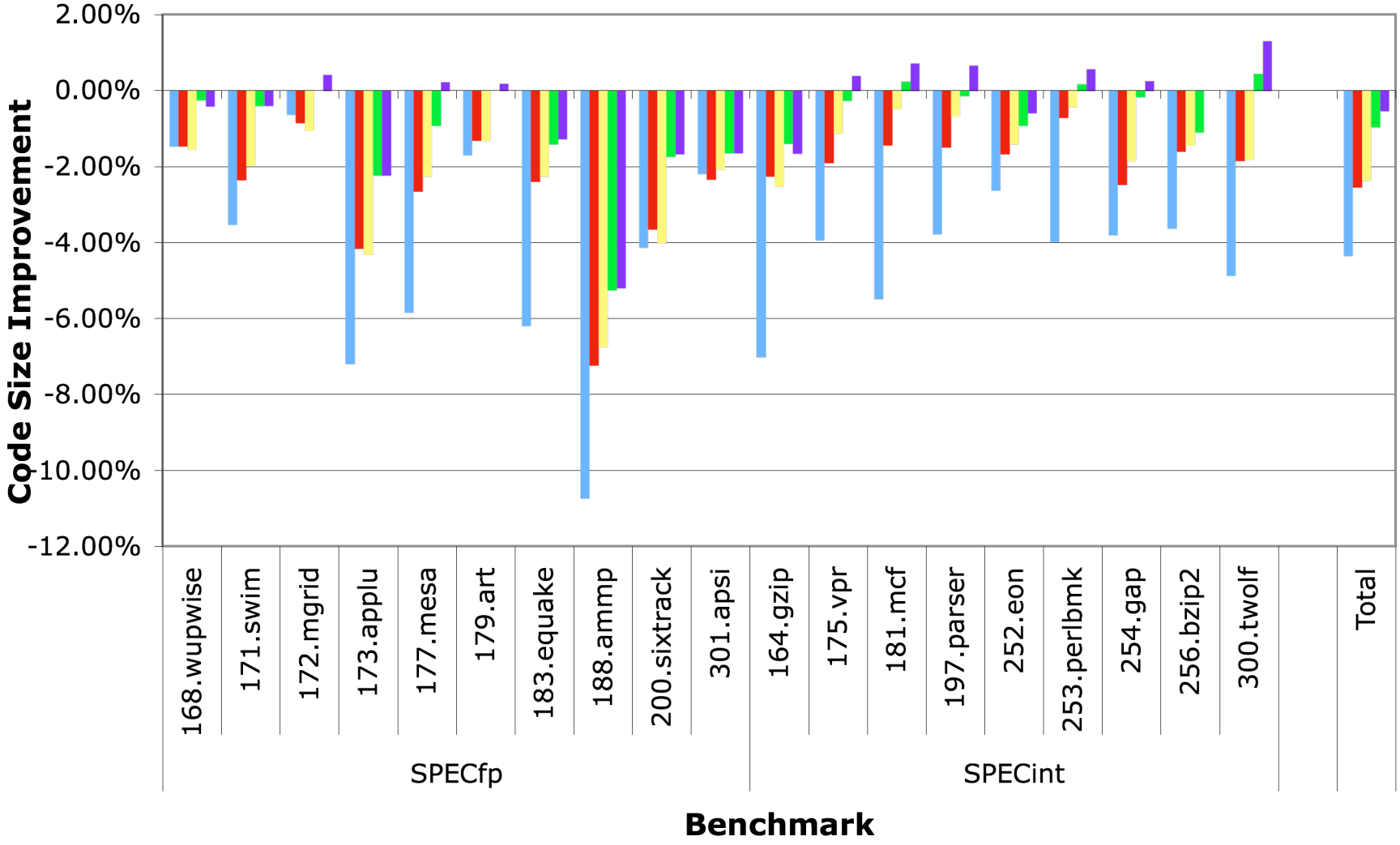
- What happens if we replace Kempe's algorithm with an optimal algorithm?



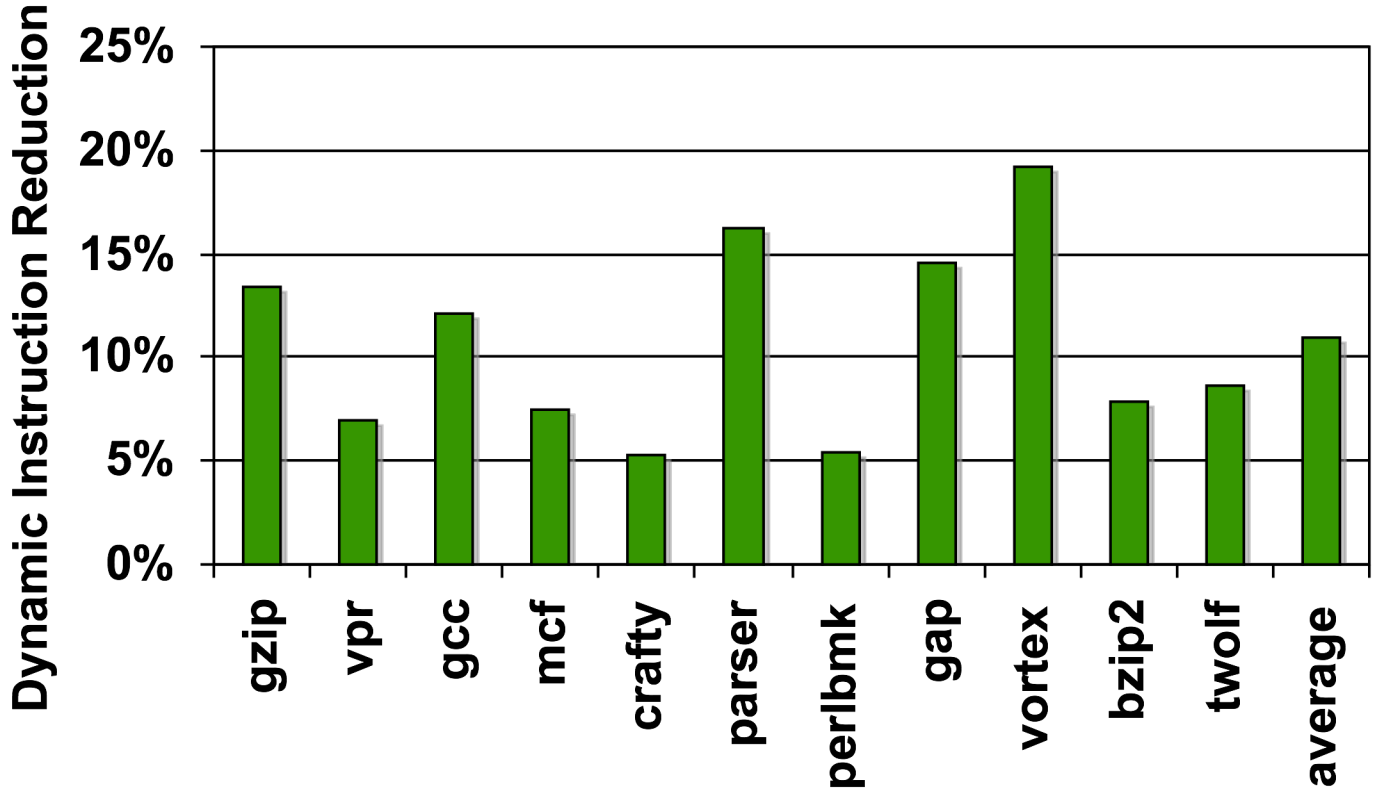
Optimal vs. Heuristic



More to register allocation than just coloring!



An optimal register allocator



Changqing Fu, Kent Wilken, and David Goodwin. A faster optimal register allocator. The Journal of Instruction-Level Parallelism, 7:1–31, January 2005.

Allocating for PA-RISC (24 registers) compare to gcc 2.5.7

Summary

- Graph coloring allocator is effective
  - coloring not that important
    - simple algorithm works well
  - subtleties are important
    - dealing with live ranges
    - what and when to spill
    - coalescing