Design of an SSA Register Allocator

SSA '09

Sebastian Hack

SAARLAND UNIVERSITY
COMPUTER SCIENCE

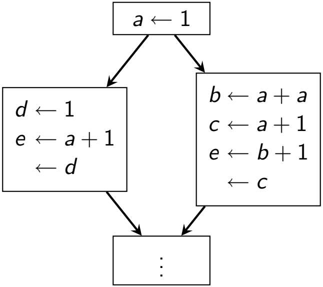
Part I

Foundations

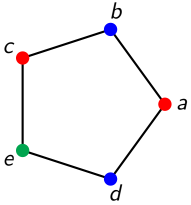
Non-SSA Interference Graphs

An inconvenient property

Program



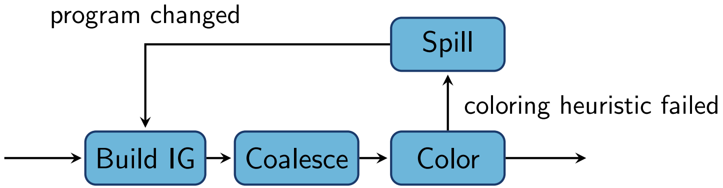
Interference Graph



- The number of live variables at each instruction (register pressure) is 2
- However, we need 3 registers for a correct register allocation
- This gap can be arbitrarily large

Graph-Coloring Register Allocation

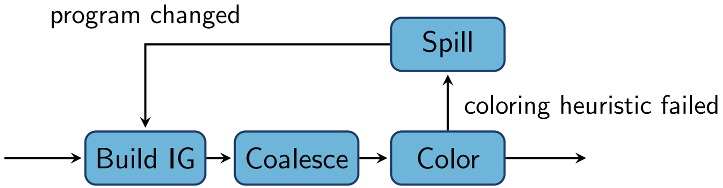
[Chaitin '80, Briggs '92, Appel & George '96, Park & Moon '04]



- Every undirected graph can occur as an interference graph
  $\Longrightarrow$ Finding a k-coloring is NP-complete
- Color using heuristic
  $\Longrightarrow$ Iteration necessary
- Might introduce spills although IG is k-colorable
- Rebuilding the IG each iteration is costly

Graph-Coloring Register Allocation

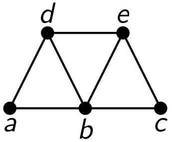
[Chaitin '80, Briggs '92, Appel & George '96, Park & Moon '04]



- Spill-code insertion is crucial for the program's performance
- Hence, it should be very sensitive to the structure of the program
  - Place load and stores carefully
  - Avoid spilling in loops!
- Here, it is merely a fail-safe for coloring

Coloring

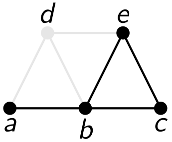
- Subsequently remove the nodes from the graph



elimination order

Coloring

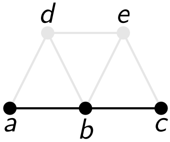
- Subsequently remove the nodes from the graph



elimination order
d,

Coloring

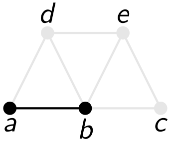
- Subsequently remove the nodes from the graph



elimination order
d, e,

Coloring

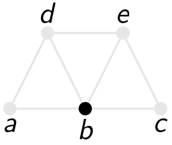
- Subsequently remove the nodes from the graph



elimination order
d, e, c,

Coloring

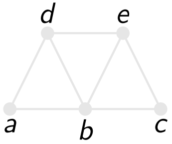
- Subsequently remove the nodes from the graph



elimination order
d, e, c, a,

Coloring

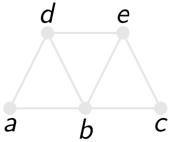
- Subsequently remove the nodes from the graph



elimination order
d, e, c, a, b

Coloring

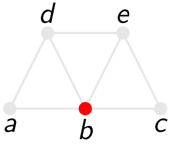
- Subsequently remove the nodes from the graph
- Re-insert the nodes in reverse order
- Assign each node the next possible color



elimination order
d, e, c, a, b

Coloring

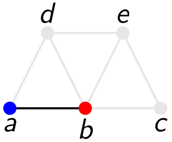
- Subsequently remove the nodes from the graph
- Re-insert the nodes in reverse order
- Assign each node the next possible color



elimination order
d, e, c, a,

Coloring

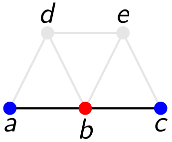
- Subsequently remove the nodes from the graph
- Re-insert the nodes in reverse order
- Assign each node the next possible color



elimination order
d, e, c,

Coloring

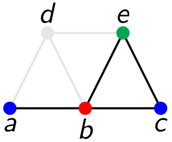
- Subsequently remove the nodes from the graph
- Re-insert the nodes in reverse order
- Assign each node the next possible color



elimination order
d, e,

Coloring

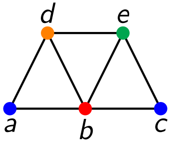
- Subsequently remove the nodes from the graph
- Re-insert the nodes in reverse order
- Assign each node the next possible color



elimination order
d,

Coloring

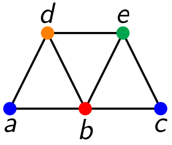
- Subsequently remove the nodes from the graph
- Re-insert the nodes in reverse order
- Assign each node the next possible color



elimination order

Coloring

- Subsequently remove the nodes from the graph
- Re-insert the nodes in reverse order
- Assign each node the next possible color



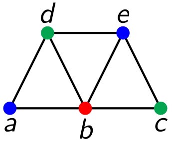
elimination order

But...

this graph is 3-colorable. We obviously picked a wrong order.

Coloring

- Subsequently remove the nodes from the graph
- Re-insert the nodes in reverse order
- Assign each node the next possible color



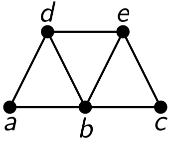
elimination order

But. . .

this graph is 3-colorable. We obviously picked a wrong order.

Perfect Elimination Order (PEO)

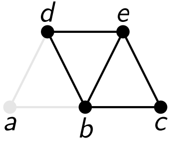
All not yet eliminated neighbors of a node are mutually connected



elimination order

Perfect Elimination Order (PEO)

All not yet eliminated neighbors of a node are mutually connected

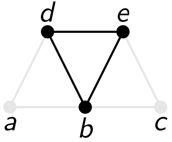


elimination order
a,

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Perfect Elimination Order (PEO)

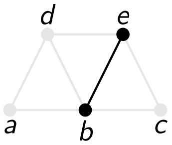
All not yet eliminated neighbors of a node are mutually connected



elimination order
a, c,

Perfect Elimination Order (PEO)

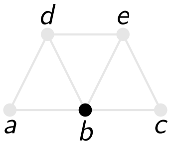
All not yet eliminated neighbors of a node are mutually connected



elimination order
a, c, d,

Perfect Elimination Order (PEO)

All not yet eliminated neighbors of a node are mutually connected

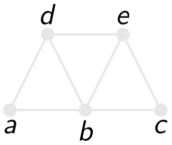


elimination order
a, c, d, e,

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Perfect Elimination Order (PEO)

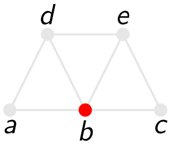
All not yet eliminated neighbors of a node are mutually connected



elimination order
a, c, d, e, b

Perfect Elimination Order (PEO)

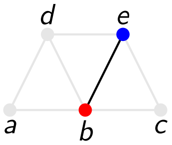
All not yet eliminated neighbors of a node are mutually connected



elimination order
a, c, d, e,

Perfect Elimination Order (PEO)

All not yet eliminated neighbors of a node are mutually connected

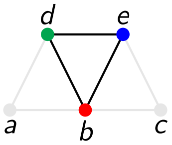


elimination order
a, c, d,

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Perfect Elimination Order (PEO)

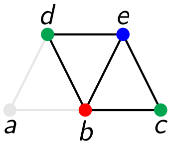
All not yet eliminated neighbors of a node are mutually connected



elimination order
a, c,

Perfect Elimination Order (PEO)

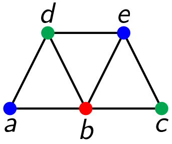
All not yet eliminated neighbors of a node are mutually connected



elimination order
a,

Perfect Elimination Order (PEO)

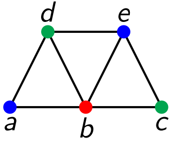
All not yet eliminated neighbors of a node are mutually connected



elimination order

Perfect Elimination Order (PEO)

All not yet eliminated neighbors of a node are mutually connected



elimination order

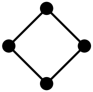
From Graph Theory [Berge '60, Fulkerson/Gross '65, Gavril '72]

- A PEO allows for an optimal coloring in $k \times |V|$
- The number of colors is bound by the size of the largest clique

Coloring

PEOs

- Graphs with holes larger than 3 have no PEO, e.g.

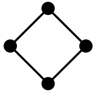


- Graphs with PEOs are called chordal

Coloring

PEOs

- Graphs with holes larger than 3 have no PEO, e.g.



- Graphs with PEOs are called chordal

Core Theorem of SSA Register Allocation
[Brisk; Bouchez, Darte, Rastello; Hack, around 2005]

- The dominance relation in SSA programs induces a PEO in the IG
- Thus, SSA IGs are chordal

Properties of SSA Register Allocation

SAARLAND UNIVERSITY
COMPUTER SCIENCE

- Before a value v is added to a PEO,
  add all values whose definitions are dominated by v
- A post order walk of the dominance tree defines a PEO
- A pre order walk of the dominance tree yields a coloring sequence
- IGs of SSA-form programs can be colored optimally in $O(k \cdot |V|)$
- Without constructing the interference graph itself
- Number of needed registers is exactly determined by register pressure
- After lowering the pressure, no additional spills will be introduced

Properties of SSA Register Allocation

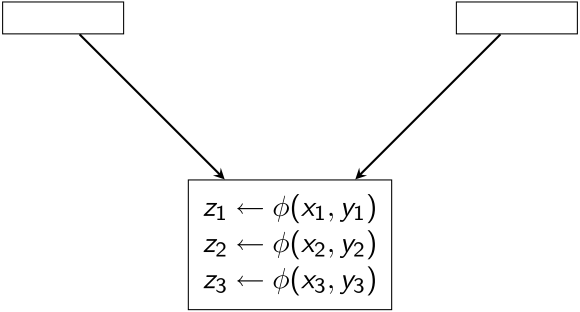
- Before a value v is added to a PEO,
  add all values whose definitions are dominated by v
- A post order walk of the dominance tree defines a PEO
- A pre order walk of the dominance tree yields a coloring sequence
- IGs of SSA-form programs can be colored optimally in $O(k \cdot |V|)$
- Without constructing the interference graph itself
- Number of needed registers is exactly determined by register pressure
- After lowering the pressure, no additional spills will be introduced

But ...

What about the ϕ-functions?

Φ-Functions

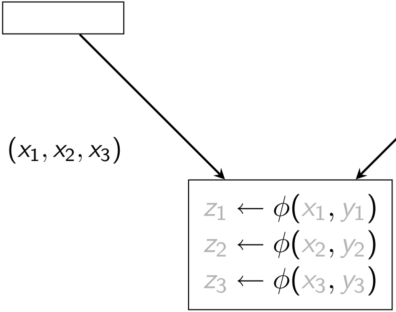
- Consider following example



$$
\begin{aligned}
z_1 &\leftarrow \phi(x_1, y_1) \\
z_2 &\leftarrow \phi(x_2, y_2) \\
z_3 &\leftarrow \phi(x_3, y_3)
\end{aligned}
$$

Φ-Functions

- Consider following example



$(z_1, z_2, z_3) \leftarrow (x_1, x_2, x_3)$

$(z_1, z_2, z_3) \leftarrow (y_1, y_2, y_3)$

$$
\begin{aligned}
z_1 &\leftarrow \phi(x_1, y_1) \\
z_2 &\leftarrow \phi(x_2, y_2) \\
z_3 &\leftarrow \phi(x_3, y_3)
\end{aligned}
$$

- Φ-functions are parallel copies on control flow edges

Φ-Functions

- Consider following example

$(z_1, z_2, z_3) \leftarrow (x_1, x_2, x_3)$

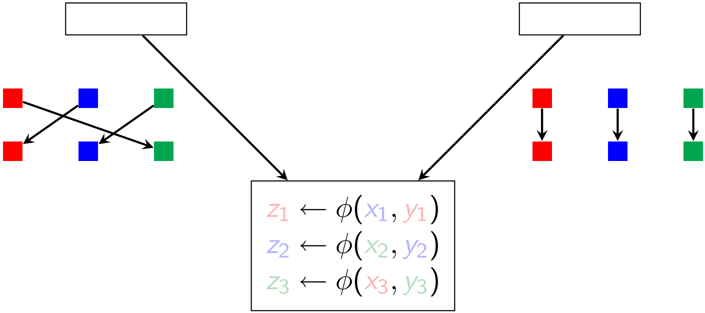
$(z_1, z_2, z_3) \leftarrow (y_1, y_2, y_3)$

$$z_1 \leftarrow \phi(x_1, y_1)$$
$$z_2 \leftarrow \phi(x_2, y_2)$$
$$z_3 \leftarrow \phi(x_3, y_3)$$

- Φ-functions are parallel copies on control flow edges
- Considering assigned registers ...

Φ-Functions

- Consider following example



- Φ-functions are parallel copies on control flow edges
- Considering assigned registers ...
- ... Φs represent register permutations

Intuition: Why are SSA IGs chordal?

Straight-line code

Program

$a \leftarrow \cdots$

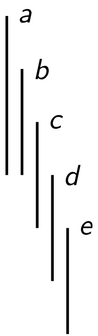
$b \leftarrow \cdots$

$c \leftarrow \cdots$

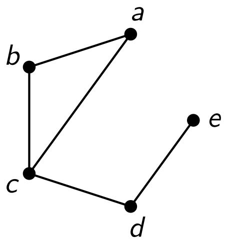
$d \leftarrow a + b$

$e \leftarrow c + 1$

Live Ranges



Interference Graph

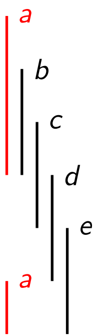


- How can we create a 4-cycle $\{a, c, d, e\}$?

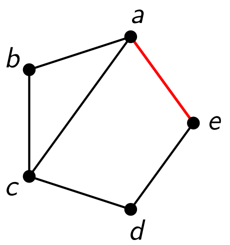
Intuition: Why are SSA IGs chordal?

Straight-line code

Program	Live Ranges
$\color{red}{a \leftarrow \cdots}$	
$b \leftarrow \cdots$	
$c \leftarrow \cdots$	
$d \leftarrow a + b$	
$e \leftarrow c + 1$	
$\color{red}{a \leftarrow \cdots}$	



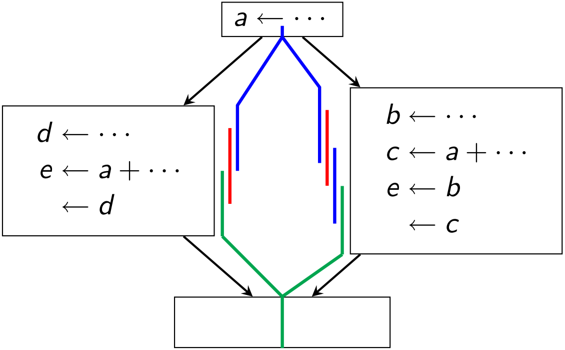
Interference Graph



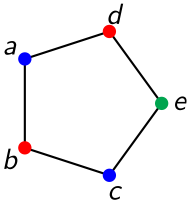
- How can we create a 4-cycle $\{a, c, d, e\}$?
- Redefine $a \implies$ SSA violated!

Intuition: ϕ-functions break cycles in the IG

Program and live ranges

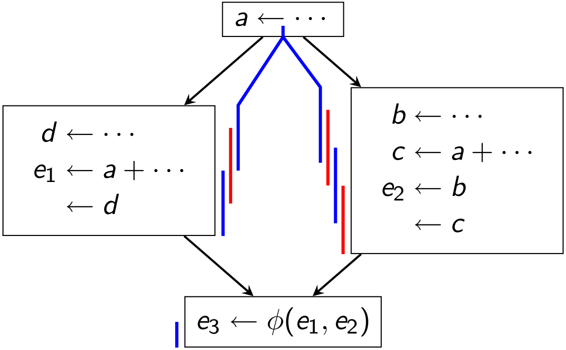


Interference Graph

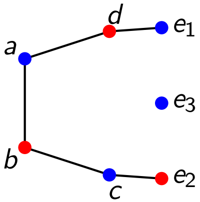


Intuition: ϕ-functions break cycles in the IG

Program and live ranges



Interference Graph



Intuition: Why Parallel Copies are Good

Parallel copies

$$(a', b', c', d') \leftarrow (a, b, c, d)$$

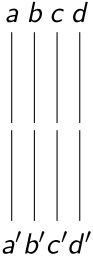
Sequential copies

$$\begin{aligned} d' &\leftarrow d \\ c' &\leftarrow c \\ b' &\leftarrow b \\ a' &\leftarrow a \end{aligned}$$

Intuition: Why Parallel Copies are Good

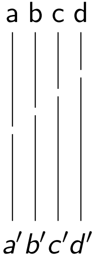
Parallel copies

$$(a', b', c', d') \leftarrow (a, b, c, d)$$



Sequential copies

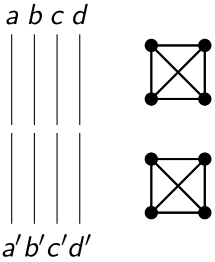
$$\begin{aligned} d' &\leftarrow d \\ c' &\leftarrow c \\ b' &\leftarrow b \\ a' &\leftarrow a \end{aligned}$$



Intuition: Why Parallel Copies are Good

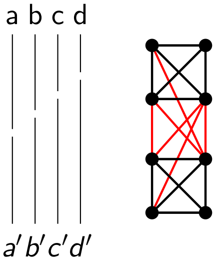
Parallel copies

$$(a', b', c', d') \leftarrow (a, b, c, d)$$



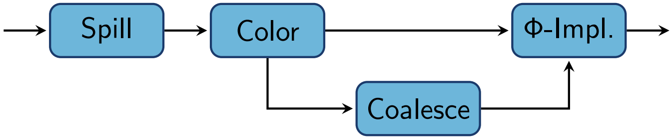
Sequential copies

$$\begin{aligned} d' &\leftarrow d \\ c' &\leftarrow c \\ b' &\leftarrow b \\ a' &\leftarrow a \end{aligned}$$



Summary so far

- IGs of SSA-form programs are chordal
- The dominance relation induces a PEO
- Architecture without iteration



- Register assignment optimal in linear time
- Do not need to construct interference graph

Part II

Register Constraints

Handling of Register Constraints

- Certain instructions require operand to reside in special register
- Instruction set architecture (ISA), e.g.:
  Shift count must be in `cl` on x86
- Calling conventions, e.g.:
  First integer argument of function in `R3` on PPC/Linux
- Caller-/Callee-save registers within a function

Usual way of handling constraints

IR:

$\cdots \leftarrow$ `call foo` t_1, t_2, t_3

Lower IR:

```
...
mov R3, t1
mov R4, t2
mov R5, t3
call foo
...
```

- Registers are like variables in the lower IR
- Multiple assignments possible (breaks SSA!)

Has poor engineering properties:

- Always special case in the code
- Does `R3` interfere with `t1`?
- How long can a reg live range be?

Even worse

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Theorem [Marx '05]

If a chordal graph contains two nodes precolored to the same color, coloring is NP-complete

Solution:

- Split all live ranges in front of the constrained instruction
- Separates graph into two components
- Annotate the constraints at the instruction
- Let the coloring algorithm fulfill the constraints
- Basically pushes the problem to the coalescer

Example

Before:

$$
\begin{aligned}
a &\leftarrow \cdots \\
&\vdots \\
&\leftarrow \texttt{call}\ \texttt{foo}\ (b, c, d) \\
&\vdots \\
&\leftarrow a
\end{aligned}
$$

After:

$$
\begin{aligned}
a &\leftarrow \cdots \\
&\vdots \\
(a', {\color{red}b'}, {\color{blue}c'}, {\color{green}d'}) &\leftarrow (a, b, c, d) \\
&\leftarrow \texttt{call}\ \texttt{foo}\ (b', c', d') \\
&\vdots \\
&\leftarrow a'
\end{aligned}
$$

Caller-/Callee-Save

- Can be modelled by normal register constraints
- Callee-Save registers are implicit parameters to a function
- Caller-Save registers are implicit results of a function
- Insert dummy SSA variables for these parameters
- The spiller will (transparently) do the rest

$$
\begin{aligned}
(c_1, c_2) &\leftarrow \texttt{start} \\
&\vdots \\
(r_1, r_2) &\leftarrow \texttt{call}\ \texttt{foo}(b, c, d) \\
&\ \texttt{dummy\_use}(r_1, r_2) \\
&\vdots \\
&\leftarrow \texttt{end}(c_1, c_2)
\end{aligned}
$$

Part III

Spilling

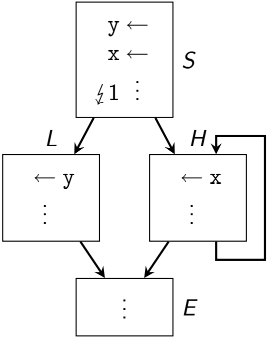
Spilling

SSA-Form Register Allocation

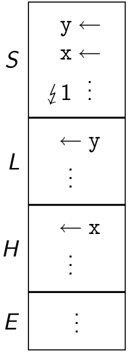
- Spilling is not dependent on the coloring algorithm
- Do not spill nodes in an interference graph
- To color optimally:
  Reduce register pressure to number of available registers
- Can insert store and load instructions sensitively to the program's structure
- Most important:
  - Pull reloads in front loops
  - Push stores behind loops
- Revisit Belady's algorithm

Linear Scan

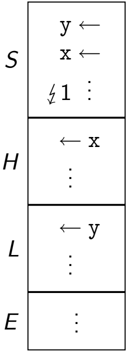
Linearizations



Example CFG



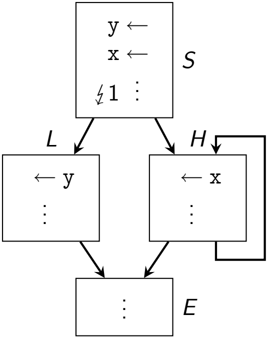
- x spilled
- Bad: Reload in loop



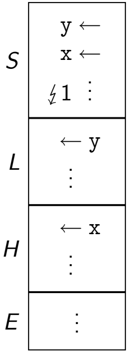
- y spilled
- Good: No reload in loop

Linear Scan

Linearizations



Example CFG



Linearization

- Register occupation at entry of *H* is given by exit of *L*!
- However, there is no control-flow between both
- Example last slide:
  - Linearization dictates reloads
  - Might unnecessarily reload in loops!
- Why do we linearize at all?

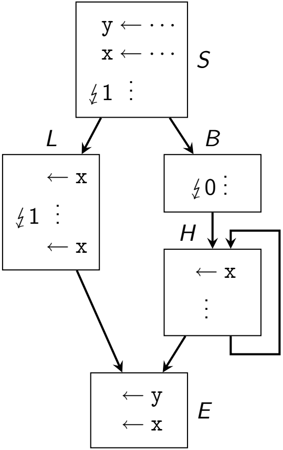
Belady on CFGs

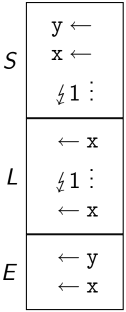
- Belady evicts the variable whose next use is farthest in the future
- Good because frees register for the longest possible time
- On straight-line code minimum number of replacements

Our goals:

- Extend Belady to CFGs
- Try to emulate Belady on each trace as good as possible
- Keep it simple: Apply Belady to each basic block once
- Where can we tweak?
  - Next-use distance
  - Occupation of the registers at entry of each block

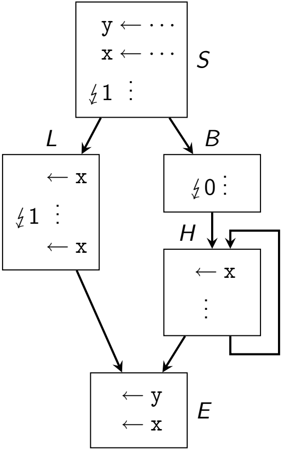
Belady on Traces





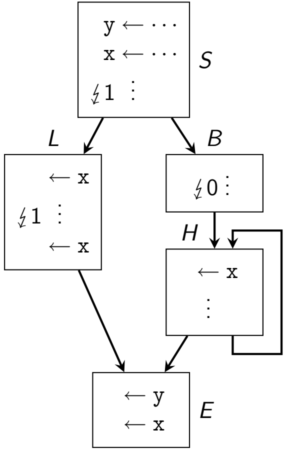
- One of x, y has to be spilled at the end of S
- Use of y is farther away
- We cannot know this by only looking at S
- Conclusion:
  Need global next-uses distances!

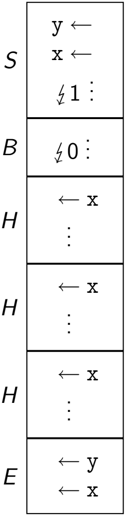
Belady on Traces



- Consider E
- x is in a register on both incoming branches
- We can assume it to be in registers on the entry of E
- Conclusion:
  Processing predecessors first makes register occupation available

Belady on Traces





- Neither x nor y can "survive" *B*
- x is reloaded in first execution of *H*
- Can be used from a register ever after
- Conclusion:
  Provide "loop workset" at loop entrances

Our Approach

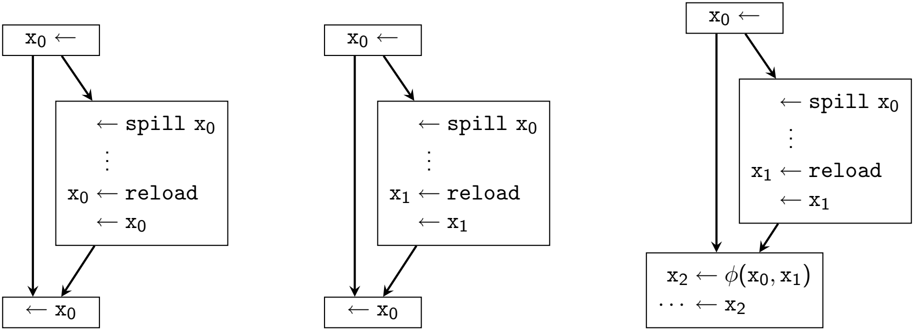
[Braun & Hack, CC'09]

- Apply furthest-first algorithm to each block in the CFG once
- Do not flatten the CFG

Algorithm

1. Compute global next uses (entails liveness!)
2. For each block B in reverse post order of the CFG:
    1. Determine initialization of register set sensitive to CF predecessors
    2. Insert coupling code at the block entry
    3. Perform Belady's algorithm on B
3. Reconstruct SSA

SSA Reconstruction



- Inserting reloads for variables creates additional definitions
- Violates SSA
- Thus, SSA has to be reconstructed after spilling
- Use algorithm by [Sastry & Ju PLDI'97]

Results

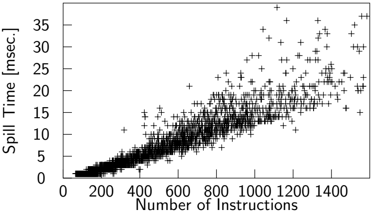
- Implemented in our x86 research compiler libFirm
- Features SSA-based register allocator
- Ran `CINT2000` benchmark
- Compare against Chaitin/Briggs graph-coloring allocator (GC)
  LLVM's linear scan (LS)

Quality

Reduction of executed spills
and reloads against:

	GC	LS
Reloads	58.2%	54.5%
Spills	41.9%	61.5%

Compilation Speed



Average throughput:
430 insns per msec
(2GHz Core 2 Duo)

Part IV

Coalescing

Coalescing

[Hack & Goos, PLDI'08]

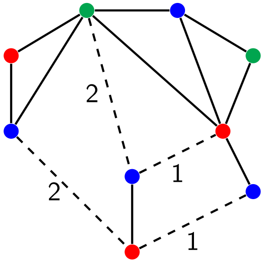
- Do not modify the graph
- Modify the coloring!
- Try to assign copy-related nodes the same color
- Introduce cost function for colorings
  $\Longrightarrow$ Sum of all weights of unfulfilled affinities

Coalescing

[Hack & Goos, PLDI'08]

- Do not modify the graph
- Modify the coloring!
- Try to assign copy-related nodes the same color
- Introduce cost function for colorings
  $\Longrightarrow$ Sum of all weights of unfulfilled affinities

Initial coloring (cost: 6)

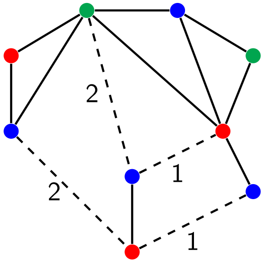


Coalescing

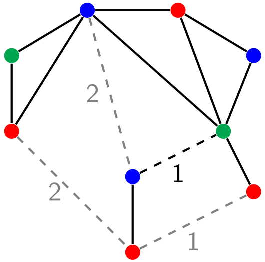
[Hack & Goos, PLDI'08]

- Do not modify the graph
- Modify the coloring!
- Try to assign copy-related nodes the same color
- Introduce cost function for colorings
  $\Longrightarrow$ Sum of all weights of unfulfilled affinities

Initial coloring (cost: 6)



Better coloring (cost: 1)



Coalescing

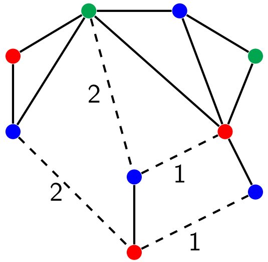
[Hack & Goos, PLDI'08]

- Do not modify the graph
- Modify the coloring!
- Try to assign copy-related nodes the same color
- Introduce cost function for colorings
  $\Longrightarrow$ Sum of all weights of unfulfilled affinities

- Coalesce after coloring

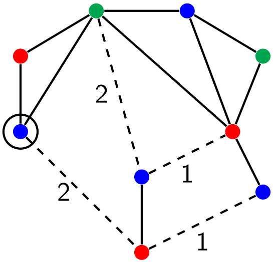
Recoloring

- Optimistically try to assign move-related nodes the same color
- Resolve color clashes recursively through the graph



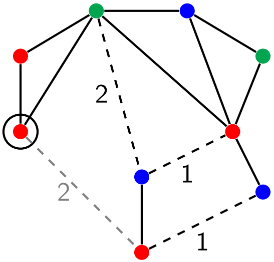
Recoloring

- *Optimistically* try to assign move-related nodes the same color
- Resolve color clashes recursively through the graph



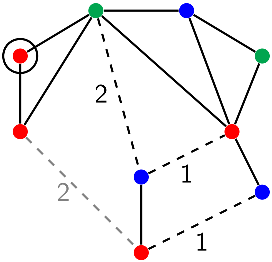
Recoloring

- Optimistically try to assign move-related nodes the same color
- Resolve color clashes recursively through the graph



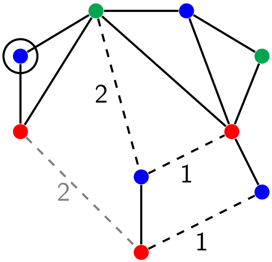
Recoloring

- Optimistically try to assign move-related nodes the same color
- Resolve color clashes recursively through the graph



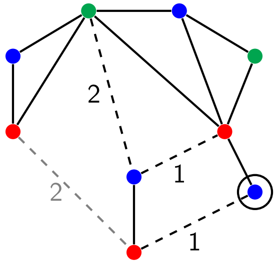
Recoloring

- Optimistically try to assign move-related nodes the same color
- Resolve color clashes recursively through the graph



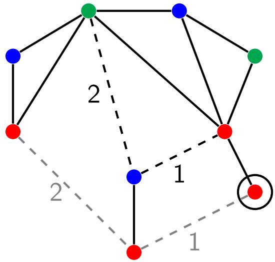
Recoloring

- Optimistically try to assign move-related nodes the same color
- Resolve color clashes recursively through the graph



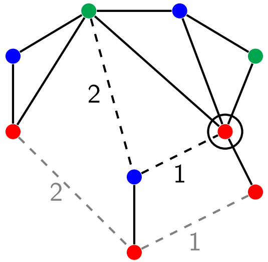
Recoloring

- Optimistically try to assign move-related nodes the same color
- Resolve color clashes recursively through the graph



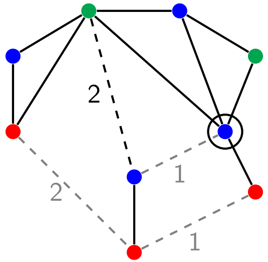
Recoloring

- Optimistically try to assign move-related nodes the same color
- Resolve color clashes recursively through the graph



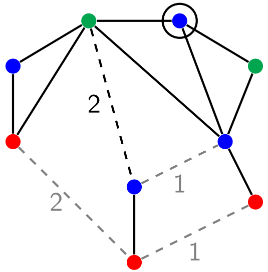
Recoloring

- Optimistically try to assign move-related nodes the same color
- Resolve color clashes recursively through the graph



Recoloring

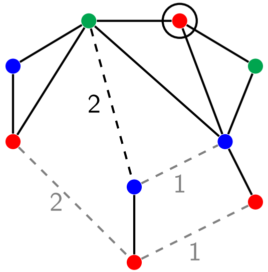
- Optimistically try to assign move-related nodes the same color
- Resolve color clashes recursively through the graph



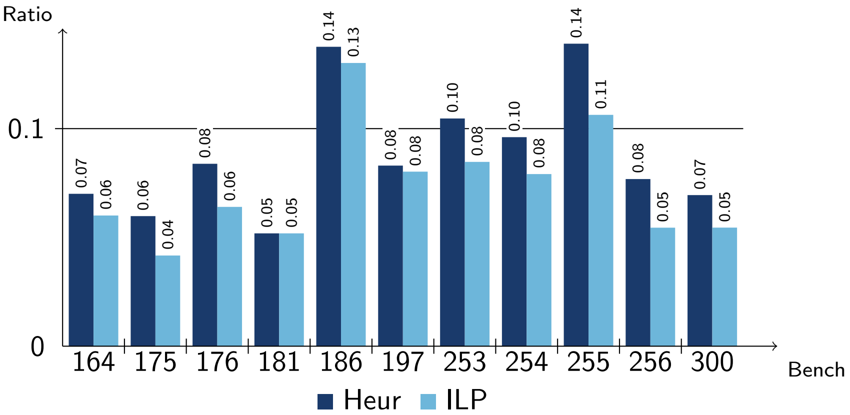
Recoloring

SAARLAND UNIVERSITY
COMPUTER SCIENCE

- Optimistically try to assign move-related nodes the same color
- Resolve color clashes recursively through the graph



Quality of the Results



geomean Heur: 0.084, geomean ILP: 0.067

Sum of weights of unfulfilled affinities after optimization relative to unoptimized

Comparison to existing techniques

SAARLAND UNIVERSITY
COMPUTER SCIENCE

- Conservative Coalescing
  - Best known conservative coalescing technique
  - Costs left over by IRC were reduced by 22.5%
  - Number of copies left over by IRC reduced by 44.3%

- Aggressive/Optimistic Coalescing
  - Did not compare to aggressive coalescing algorithms
  - May spill $\implies$ different problem

Conclusions

- Coloring is easy
- SSA separates spilling from coalescing
  $\Longrightarrow$ Simplifies engineering
- Both remain hard and challenging
- Spilling can be more sensitive to program
  $\Longrightarrow$ no additional spills due to failed coloring
- Coalescing never violates the coloring
- We never insert a spill/reload in favor of a saved copy

Conclusions

- Coloring is easy
- SSA separates spilling from coalescing
  $\Longrightarrow$ Simplifies engineering
- Both remain hard and challenging
- Spilling can be more sensitive to program
  $\Longrightarrow$ no additional spills due to failed coloring
- Coalescing never violates the coloring
- We never insert a spill/reload in favor of a saved copy
- Everything implemented within

  `http://www.libfirm.org`

  and is more than a proof of concept:
  Our Quake server is compiled with libFirm ;)
- Michael Beck will present libFirm on Thursday

Runtime of the Algorithm