

# Register Allocation for SSA Form Programs

Sebastian Hack<sup>1</sup>, Daniel Grund<sup>2</sup>, Gerhard Goos<sup>3</sup>

<sup>1</sup>ENS Lyon

<sup>2</sup>Saarland University

<sup>3</sup>University of Karlsruhe

Joint work started at Karlsruhe

26.04.2007



# Outline

- 1 Preliminaries
- 2 Classical Register Allocation
- 3 SSA-Form and Register Allocation
  - SSA, Dominance and PEOs
  - Main Results
  - Proceeding
- 4 Coalescing, Live-Range Splitting, Colorability
- 5 Summary



# Register Allocation

- Register Allocation is the task of mapping the program's variables to processor registers
- Issues to be covered:
  - **Spilling** Put variables into memory if there are not enough registers
  - **Coalescing** Eliminate unnecessary copies in the program
- Often reduced to graph coloring



## Definition

A variable  $v$  is live at a label  $\ell$  if there is a path from  $\ell$  to a use of  $v$  not containing a definition of  $v$ .

## Definition

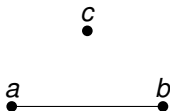
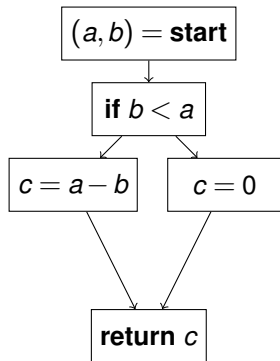
The live-range of a variable  $v$  are the labels where  $v$  is live.

- Conservative approximation by dataflow analyses



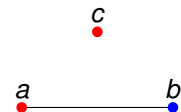
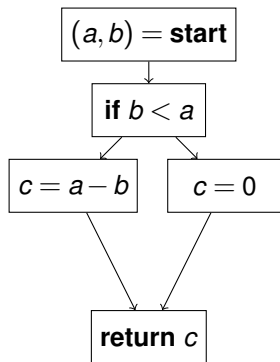
# Interference Graphs

- Two variables **interfere** if they are live at the same label
- Each variable corresponds to a node in the interference graph (IG)
- Whenever two variables interfere, there is an edge between the corresponding nodes



# Interference Graphs

- Two variables **interfere** if they are live at the same label
- Each variable corresponds to a node in the interference graph (IG)
- Whenever two variables interfere, there is an edge between the corresponding nodes



Coloring gives register allocation



# Spilling

- Register need larger than number of registers ( $\chi(IG) > k$ )
- Determine subset of (sub-)live-ranges
- Rewrite program, inserting spills and reloads to temporarily store values in memory
- Assign storage locations for spilled values



# Coalescing

- In some phases compilers insert copy instructions to
  - ▶ implement operations by code sequences (correctness)
  - ▶ simplify a translation step (engineering)
- Coalescing removes copy instructions by joining live-ranges
- Joining means assigning the same register to the source and target

```
mov ebx, [esi]          mov eax, [esi]
add ebx, ebx            add eax, eax
mov eax, ebx           ⇒
mul ecx                mul ecx
```

- More general: reduce useless value movement

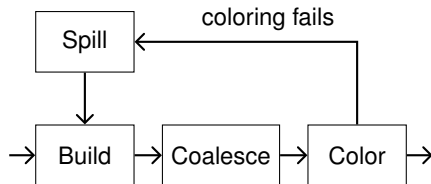


# Outline

- 1 Preliminaries
- 2 Classical Register Allocation**
- 3 SSA-Form and Register Allocation
  - SSA, Dominance and PEOs
  - Main Results
  - Proceeding
- 4 Coalescing, Live-Range Splitting, Colorability
- 5 Summary



# Chaitin/Briggs Register Allocator

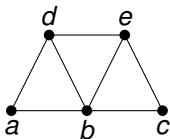


- Every undirected graph can occur as an interference graph
- Determining chromatic number is  $\mathcal{NP}$ -complete
- Color using heuristic  $\Rightarrow$  Iteration necessary on failure



# Coloring

- Subsequently remove the nodes from the graph (simplify)

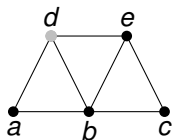


elimination order



# Coloring

- Subsequently remove the nodes from the graph (simplify)



elimination order

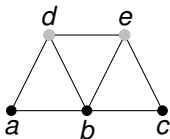
---

$d,$



# Coloring

- Subsequently remove the nodes from the graph (simplify)



elimination order

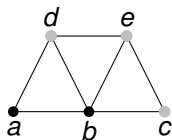
---

$d, e,$



# Coloring

- Subsequently remove the nodes from the graph (simplify)



elimination order

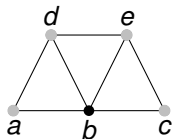
---

*d*, *e*, *c*,



# Coloring

- Subsequently remove the nodes from the graph (simplify)



elimination order

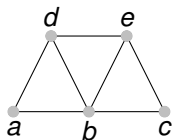
---

$d, e, c, a,$



# Coloring

- Subsequently remove the nodes from the graph (simplify)



elimination order

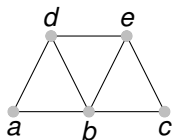
---

$d, e, c, a, b$



# Coloring

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



elimination order

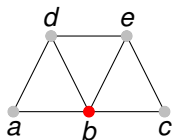
---

d, e, c, a, b



# Coloring

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



elimination order

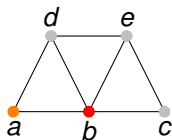
---

d, e, c, a,



# Coloring

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



elimination order

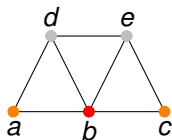
---

*d*, *e*, *c*,



# Coloring

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



elimination order

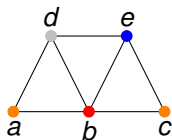
---

d, e,



# Coloring

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



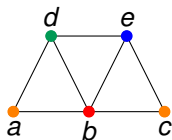
elimination order

d,



# Coloring

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



elimination order



# Spilling

- Nodes with less than  $k$  neighbors can always be colored
- Simplify phase removes those nodes
- Other nodes get spilled pessimistically

Or:

- Other nodes are optimistically removed as well
- Hope for a free color despite many neighbors
- Delays spill decision until actual color assignment

Inserting spill code invalidates liveness information  
⇒ restart by rebuilding the IG

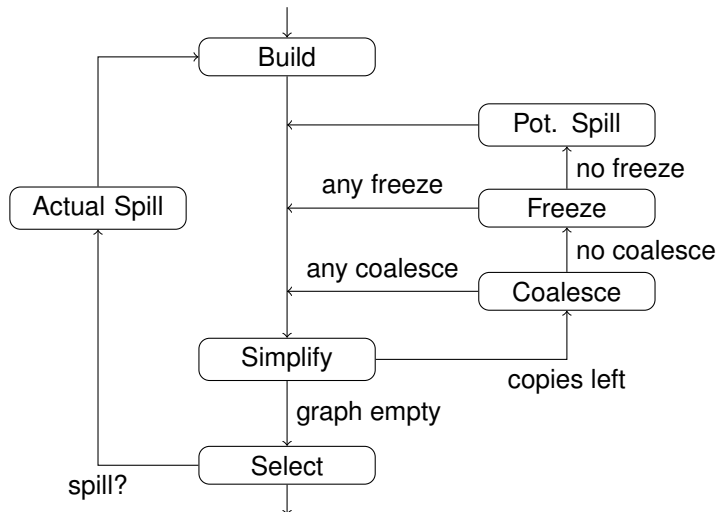


# Coalescing

- Coalescing merges nodes in the graph to force both values into the same register
- Less but longer live ranges
- Problem: Coalescing influences colorability
  
- First heuristic ignored negative effects (aggressive)
- Later approaches restricted coalescing (conservative/iterated)
- State of the art has undo-capabilities (optimistic)



# “Iterated” Allocator (Appel & George 96)



# Outline

- 1 Preliminaries
- 2 Classical Register Allocation
- 3 SSA-Form and Register Allocation**
  - SSA, Dominance and PEOs
  - Main Results
  - Proceeding
- 4 Coalescing, Live-Range Splitting, Colorability
- 5 Summary



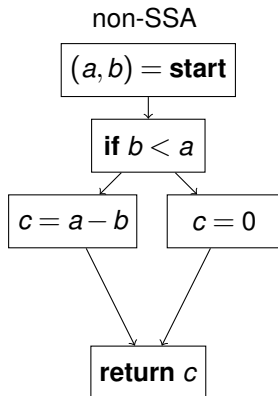
# Outline

- 1 Preliminaries
- 2 Classical Register Allocation
- 3 SSA-Form and Register Allocation**
  - SSA, Dominance and PEOs
    - Main Results
    - Proceeding
- 4 Coalescing, Live-Range Splitting, Colorability
- 5 Summary



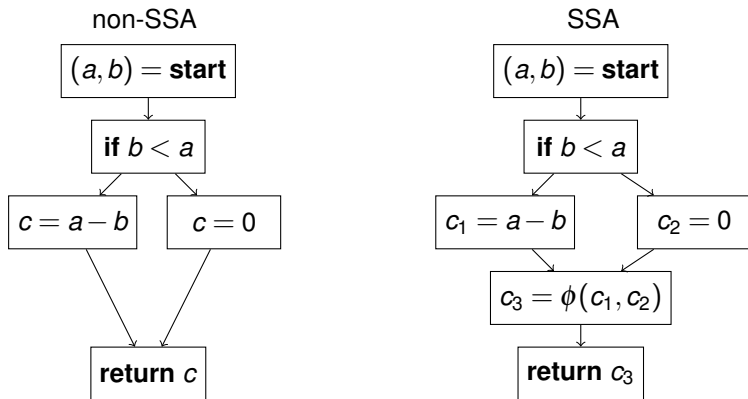
# Static Single Assignment Form

- Each variable has exactly one definition  
⇒ Identity of variables and dynamic constants (values)



# Static Single Assignment Form

- Each variable has exactly one definition  
⇒ Identity of variables and dynamic constants (values)
- $\phi$ -operations select values dependent on control flow

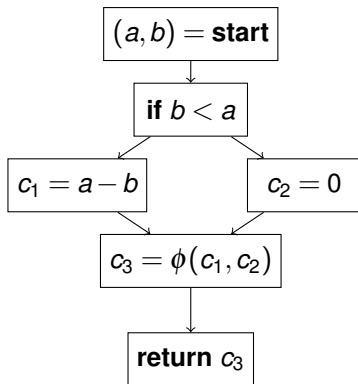


# Dominance

Crucial for SSA-form programs is the concept of dominance:

## Definition

$l_1$  dominates  $l_2$  if each path from **start** to  $l_2$  goes through  $l_1$



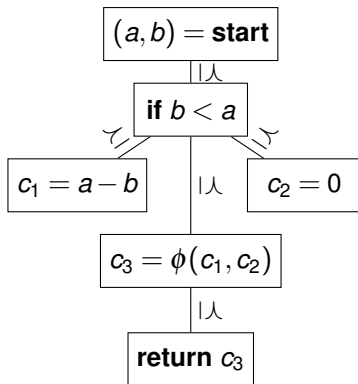
# Dominance

Crucial for SSA-form programs is the concept of dominance:

## Definition

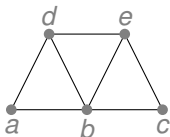
$l_1$  dominates  $l_2$  if each path from **start** to  $l_2$  goes through  $l_1$

- Each node has a unique *immediate dominator*
- Thus, dominance induces a tree on the control flow graph
- Thus, dominance is also a partial order



# Perfect Elimination Orders

- Restriction: To remove a node  $n$ , all neighbors must form a clique



elimination order

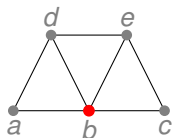
---

$a, c, d, e, b$



# Perfect Elimination Orders

- Restriction: To remove a node  $n$ , all neighbors must form a clique



elimination order

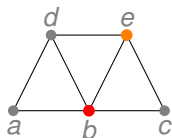
---

$a, c, d, e,$



# Perfect Elimination Orders

- Restriction: To remove a node  $n$ , all neighbors must form a clique



elimination order

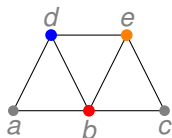
---

$a, c, d,$



# Perfect Elimination Orders

- Restriction: To remove a node  $n$ , all neighbors must form a clique



elimination order

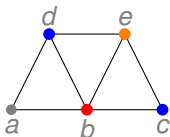
---

$a, c,$



# Perfect Elimination Orders

- Restriction: To remove a node  $n$ , all neighbors must form a clique



elimination order

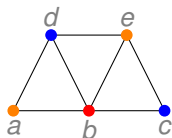
---

$a,$



# Perfect Elimination Orders

- Restriction: To remove a node  $n$ , all neighbors must form a clique



elimination order



## Perfect Elimination Orders II

- Not every graph has a PEO, e.g.



- The graphs that have PEOs are exactly the class of *chordal* graphs



## Perfect Elimination Orders II

- Not every graph has a PEO, e.g.



- The graphs that have PEOs are exactly the class of *chordal* graphs

### Definition

A graph is chordal,  
if each cycle with at least four nodes contains a chord.

### Theorem

- *Chordal graphs can be optimally colored in  $O(\omega(G) \cdot |V|)$*
- *Number of colors is bounded by the size  $\omega(G)$  of the largest clique*



# Outline

- 1 Preliminaries
- 2 Classical Register Allocation
- 3 SSA-Form and Register Allocation**
  - SSA, Dominance and PEOs
  - Main Results**
  - Proceeding
- 4 Coalescing, Live-Range Splitting, Colorability
- 5 Summary



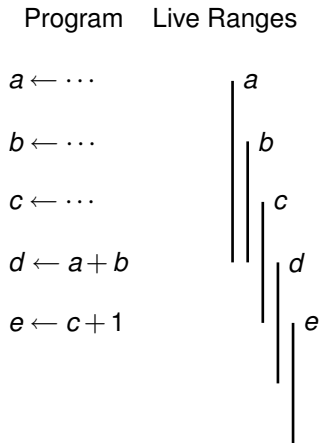
# Main Results / Driving Force

- Interference graphs of SSA-form programs are chordal
- Dominance relation induces a PEO in the interference graph of the program
- Optimal assignment of registers in  $O(\omega(G) \cdot |V|)$  without constructing the graph itself

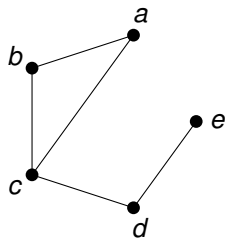
## Architecture without iteration



# Why are SSA IGs chordal? — Intuition



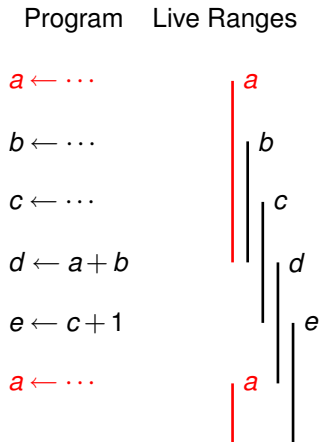
Interference Graph



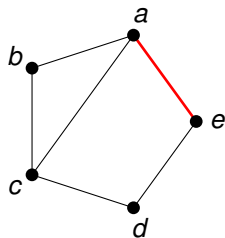
- How can we create a 4-cycle  $\{a, c, d, e\}$ ?



# Why are SSA IGs chordal? — Intuition



Interference Graph



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



# Outline

- 1 Preliminaries
- 2 Classical Register Allocation
- 3 SSA-Form and Register Allocation**
  - SSA, Dominance and PEOs
  - Main Results
  - Proceeding**
- 4 Coalescing, Live-Range Splitting, Colorability
- 5 Summary



# Spilling

- For any graph  $G$ :  $\chi(G) \geq \omega(G)$
- For chordal graphs:  $\chi(G) = \omega(G)$

## Theorem

For each clique in the IG there is a label in the program where all nodes in the clique are live.

- $\chi(IG)$  is **exactly** determined by the size of the live sets of the labels
- Lowering the number of values live at each label to  $k$  makes the IG  $k$ -colorable
- We know in advance where values must be spilled  
 $\Rightarrow$  All labels where the pressure is larger than  $k$



# Coloring

- Computing the PEO explicitly is unnecessary
- Compute liveness information
- Process basic blocks in dominance order
- After spilling  $\omega(G) \leq k$
- Thus, coloring is very simple and fast  $O(n)$



# Coalescing

- Coalescing is optional but important

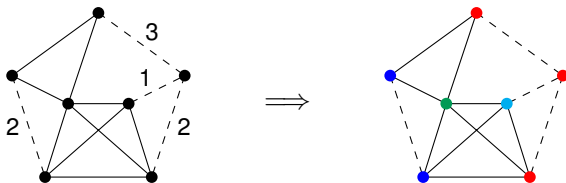


- Is an isolated subordinate optimization
  - ▶ must keep register pressure below  $k$
- Merging nodes could destroy chordality
- Information about  $\chi(IG)$  would be “lost”



# Modeling Coalescing

- Augment the interference graph by **affinity edges** representing costs of copies
- Costs are incurred if affine nodes have different colors
- Change a given coloring to reduce costs or
- find a correct coloring that minimizes the costs

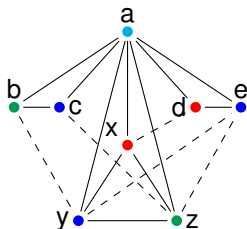
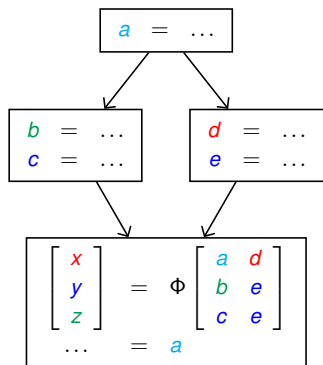


# Outline

- 1 Preliminaries
- 2 Classical Register Allocation
- 3 SSA-Form and Register Allocation
  - SSA, Dominance and PEOs
  - Main Results
  - Proceeding
- 4 Coalescing, Live-Range Splitting, Colorability
- 5 Summary



# $\Phi$ -Function Handling

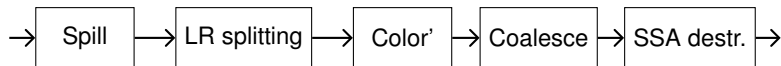


- SSA construction inserts  $\Phi$  functions
- This can be seen as live-range splitting
- Costs are modeled by affinity edges between each  $\Phi$  result and its arguments



# Register Constraint Handling (1)

- Register constraints correspond to a partial pre-coloring of the  $IG$
- This destroys equality  $\chi(G) = \omega(G)$  even for chordal graphs
- Live-range splitting can re-establish this property
- Actual problem/complexity is shifted to coalescing



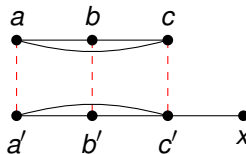
## Register Constraint Handling (2)

$$\begin{aligned}x &= op(a, b) \\ d &= c + x\end{aligned}$$



- Insert a **parallel copy** before each constrained operation  $op$

$$\begin{aligned}(a', b', c') &= (a, b, c) \\ x &= op(a', b') \\ d &= c' + x\end{aligned}$$



- This live range splitting
  - ▶ maintains  $\chi(G) = \omega(G)$
  - ▶ introduces additional affinity edges



# Live-Range Splitting and Colorability

- Chaitin:  $\forall G \exists \text{program } P : IG(P) = G$
- Live-range splitting enables efficient coloring  
SSA construction  $\Rightarrow$  chordal Graphs  
Constraint handling  $\Rightarrow$  chordal, pre-colored Graphs



## Two Similar Problems

### A) Optimal Coalescing (as above)

For a given program find a coloring such that the costs for copies are minimized

### B) Optimal Live-Range Splitting

For a given program find a set of split points and a coloring such that the costs for copies are minimized

- There are sets of split points such that a solution of A is a solution of B
- Characterization of such sets that are minimal is an open problem
- Splitting everywhere is a superset  
but this bloats the coalescing problems significantly



# Outline

- 1 Preliminaries
- 2 Classical Register Allocation
- 3 SSA-Form and Register Allocation
  - SSA, Dominance and PEOs
  - Main Results
  - Proceeding
- 4 Coalescing, Live-Range Splitting, Colorability
- 5 Summary



# Coloring

## Classical:

- Interference graph centric
- Coloring based on elimination orders
- Heuristics guide node selection

## SSA:

- Interference graph is not necessary
- Coloring based on perfect elimination orders (PEO)
- Liveness and dominance information suffice



# Spilling

## Classical:

- No possible color for node → spill decision
  - Node weights are used to associate spill-costs with nodes
  - Restart necessary after spilling
- ⇒ Coloring enabler

## SSA:

- Decisions can be based on program structure
  - Only executed once
- ⇒ Program transformation reducing local register pressures to  $k$



# Coalescing

Classical:

- Merging of nodes in the IG
- Colorability of IG may suffer
- More and more complicated techniques to avoid harmful cases

SSA:

- No merging to preserve graph class and knowledge about  $\chi(IG)$
- Expressed as an optimization problem to assign node pairs the same colors
- Strong connection to live-range splitting
- Handles complexity of register constraints as well



# Summary

- Live-range splitting enables efficient coloring schemes
- Knowledge about  $\chi(IG)$  allows sensible decoupling of subphases
- Faster allocators: no iteration, (possibly) no IG construction

