Graph-Coloring Register Allocation for Irregular Architectures

Johan Runeson &
Sven-Olof Nyström,
Uppsala Universitet

johan.runeson@it.uu.se

UPPSALA UNIVERSITET

Summary

- We have generalized Chaitin's graph-coloring global register allocation algorithm
  - Handles irregular architectures
  - Automatically retargetable
  - As fast as Chaitin's algorithm
  - Provably correct
  - Works with well-known extensions like optimistic coloring and conservative coalescing

UPPSALA UNIVERSITET

Context

- Compiler for embedded systems
  - Many (30+) different targets
    - ➔Requires retargetable algorithms
  - Irregular architectures
    - ➔Algorithms that handle irregularities
  - Large applications to compile
    - ➔Algorithms should be fast (near linear time)
  - Resource-limited target systems
    - ➔Requires high-quality output code

UPPSALA UNIVERSITET

Register Allocation

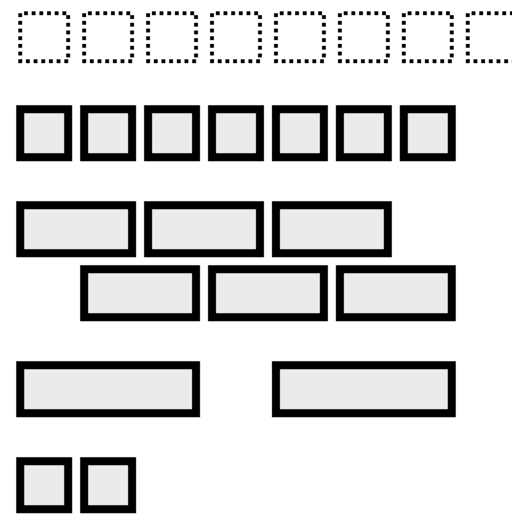
- Compiler initially assumes an infinite number of registers for *variables*
- *Register allocation* fits variables in actual registers, or *spills* to memory
  - Goal: minimize cost of spills and copies
  - Must respect constraints imposed by target architecture or runtime system
- Chaitin's global "graph-coloring" algorithm is widely used, fast, and produces high-quality allocations

UPPSALA UNIVERSITET

Irregular Architectures

- *Regular*
  - Single bank of general purpose registers

- *Irregular*
  - Many different kinds of registers
  - Resource conflicts between registers
  - Special-purpose registers



Target Characterization

- A set of *registers* (names), where registers may overlap
- A *conflict* relation determines when two registers can not be allocated at the same time
- *Register classes* restrict what registers may be assigned to a variable

UPPSALA UNIVERSITET

Graph Coloring (1)

- A variable is *live* if the value it holds may be used before it is changed
- Two variables which are live at the same time *interfere*
  - Interfering variables can not be allocated to conflicting registers
- Captured in an *interference graph*
  - A node for each variable in a function
  - Edges between interfering nodes
  - Nodes annotated with register class

Graph Coloring (2)

- An *assignment* maps each node to a register, respecting the class of the node
- An assignment is a *coloring* if it never maps two neighboring (interfering) nodes to conflicting registers
- A coloring for an interference graph is a solution to the register allocation problem

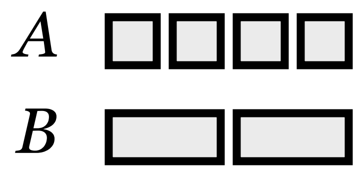
Graph Coloring (3)

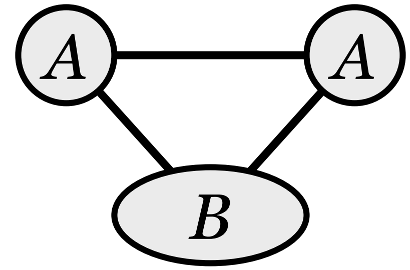
- Unfortunately, not all interference graphs can be colored
  - Register allocator may have to spill some variables to memory
- Finding (if there is) a solution is an NP-complete problem
  - May require exponential time (unless P=NP)
  - Heuristics are used in practice for fast algorithms

Coloring by Simplification

- A node is *locally colorable* if, regardless of how we assign registers to its neighbors, there is always a free register for it
- The coloring problem is *simplified* by removing a locally colorable node
  - Given a coloring for the rest of the graph, there is always a free register to assign to the node
  - Simplify recursively until the graph is empty; color nodes in reverse order

Algorithm in Action (1)



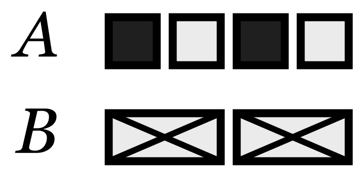


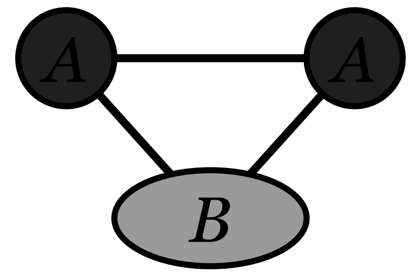
- Consider the following target architecture (left)
  - Class A has 4 registers (*a0-a3)*
  - Class B forms pairs (b0, b1) from A
- Consider an interference graph for this architecture (right)
  - Nodes annotated with class

Algorithm in Action (2)

- In the example, the A nodes are locally colorable, the B node is not

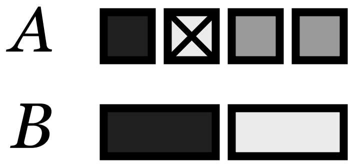
Algorithm in Action (2)

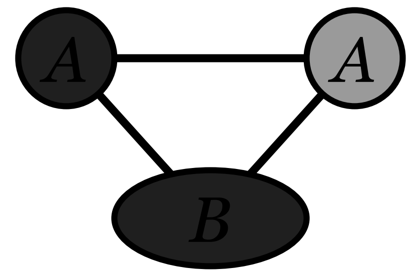




- In the example, the A nodes are locally colorable, the B node is not
  - We can assign two A:s so that the B can not be colored

Algorithm in Action (2)





- In the example, the A nodes are locally colorable, the B node is not
  - We can assign two A:s so that the B can not be colored
  - No matter how we assign one A and one B, the remaining A can be colored

Algorithm in Action (2)

- In the example, the A nodes are locally colorable, the B node is not
  - We can assign two A:s so that the B can not be colored
  - No matter how we assign one A and one B, the remaining A can be colored

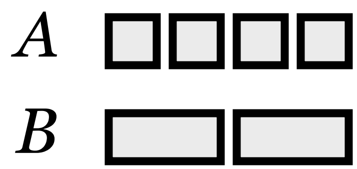
Algorithm in Action (2)

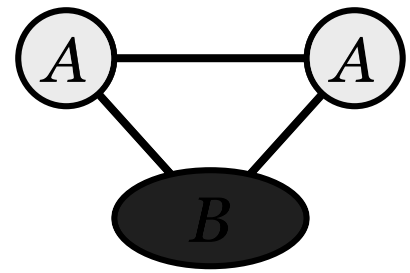
- In the example, the A nodes are locally colorable, the B node is not
  - We can assign two A:s so that the B can not be colored
  - No matter how we assign one A and one B, the remaining A can be colored

Algorithm in Action (2)

- In the example, the A nodes are locally colorable, the B node is not
  - We can assign two A:s so that the B can not be colored
  - No matter how we assign one A and one B, the remaining A can be colored

Algorithm in Action (3)





- So, the two A nodes are locally colorable, but the B node is not
- Question: Does this mean we have to spill the B?

Algorithm in Action (3)

- So, the two A nodes are locally colorable, but the B node is not
- Question: Does this mean we have to spill the B?
- Answer: No! We can still simplify the graph by removing an A

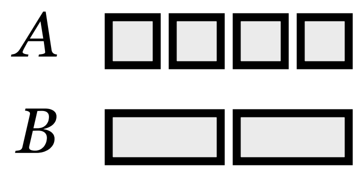
Algorithm in Action (3)

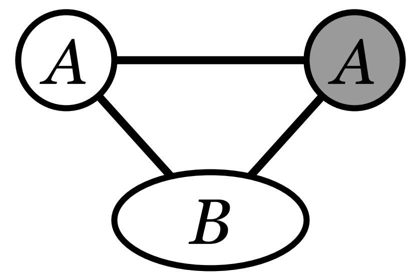
- So, the two A nodes are locally colorable, but the B node is not
- Question: Does this mean we have to spill the B?
- Answer: No! We can still simplify the graph by removing an A

Algorithm in Action (4)

- Having removed an A node, both the B node and the remaining A node are locally colorable
- Remove the B...

Algorithm in Action (4)





- Now, both the B node and the remaining A node are locally colorable
- Remove the B...
- ... and the A

Algorithm in Action (4)

- Now, both the B node and the remaining A node are locally colorable
- Remove the B...
- ... and the A
- Great! Since we could remove all nodes, we know there is a solution

Algorithm in Action (5)

- We reinsert the nodes, in reverse order of removal, and assign colors

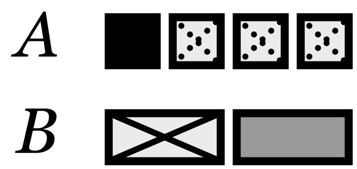
Algorithm in Action (5)

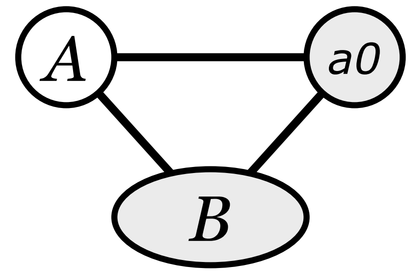
- We reinsert the nodes, in reverse order of removal, and assign colors
- First the left A node ...

Algorithm in Action (5)

- We reinsert the nodes, in reverse order of removal, and assign colors
- First the left A node gets *a0*

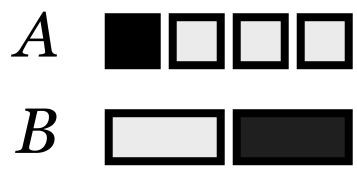
Algorithm in Action (5)

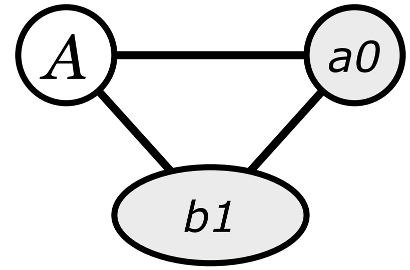




- We reinsert the nodes, in reverse order of removal, and assign colors
- First the left A node gets *a0*
- The B node interferes with the A ...

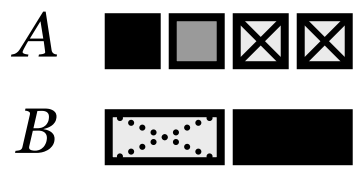
Algorithm in Action (5)

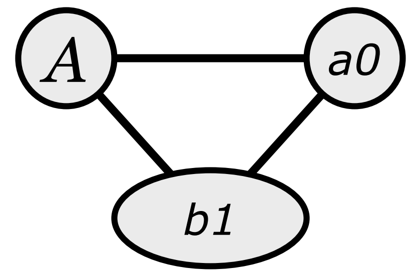




- We reinsert the nodes, in reverse order of removal, and assign colors
- First the left A node gets *a0*
- The B node interferes with the A so it must get *b1*

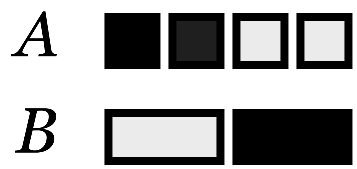
Algorithm in Action (5)

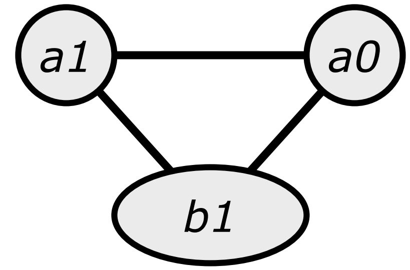




- We reinsert the nodes, in reverse order of removal, and assign colors
- First the left A node gets *a0*
- The B node interferes with the A so it must get *b1*
- The final A node interferes with both …

Algorithm in Action (5)





- We reinsert the nodes, in reverse order of removal, and assign colors
- First the left A node gets *a0*
- The B node interferes with the A so it must get *b1*
- The final A node interferes with both and thus gets *a1*

Chaitin's Algorithm

- Target characterized only by the number of registers, *k*
  - Assumes single bank of general-purpose registers
- Simplify by removing nodes with degree<*k*
- For a regular architecture, degree<*k* implies local colorability
  - Actually, they are equivalent

Degree<*k* in General

- In the example, all three nodes have the same degree, but while the A nodes are locally colorable, the B node is not
- Ergo, for an irregular architecture, degree<k does not in general imply local colorability

Precise Local Colorability

UPPSALA UNIVERSITET

- A precise test for local colorability is expensive
  - Generate and test an exponential number of possible assignments?
- The simplification algorithm works with a safe approximation, i.e. a test which implies local colorability
  - Degree<*k* is not a safe approximation (in general)

The ⟨*p*,*q*⟩ test

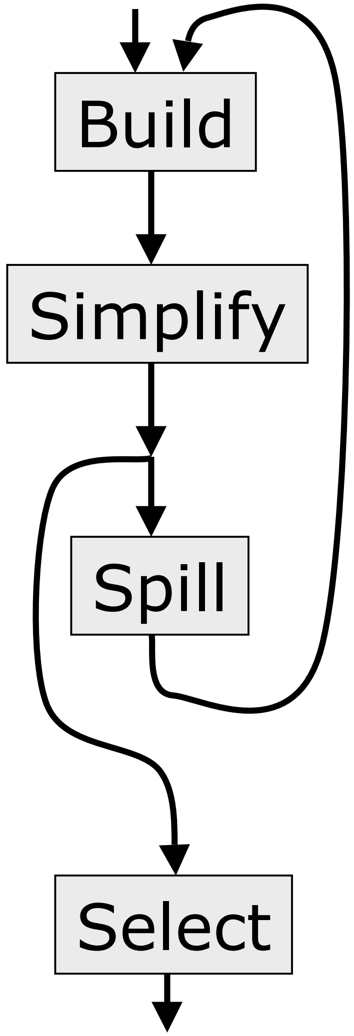
- The *⟨p,q⟩ test* safely approximates local colorability for any target
  - *p(A)* = number of registers in class *A*
  - *q(A,B)* = maximum number of registers in *A* that conflict with any single register in *B*
  - A node *n* is locally colorable if

$$\sum_{\text{neighbor } j} q(class(n), class(j)) < p(class(n))$$

Properties of ⟨*p*,*q*⟩ test

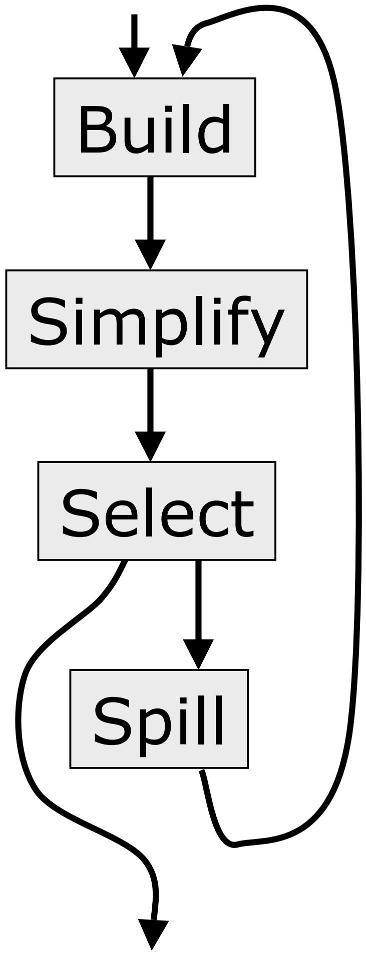
- We prove that the approximation is safe for any target (see paper)
- Compute *p* and *q* offline once per target
- Compute all ⟨*p*,*q*⟩ tests in time $O(E)$, E = number of edges
- For a regular architecture, $p = k$ and $q = 1$, so the ⟨*p*,*q*⟩ test degenerates to degree<*k*

The Complete Algorithm



- Construct interference graph
- Repeatedly remove nodes which pass the ⟨*p*,*q*⟩ test
- If the graph is non-empty, pick some node and spill it; restart from Build
- If graph is empty, reinsert nodes in reverse order of removal and assign registers

... with Optimistic Coloring



- Postpone spilling decisions from Simplify to Select
  - [Briggs et al.]
  - Simplify removes nodes optimistically instead of spilling
  - Only if Select fails to color an optimistically removed node is spilling necessary
  - Other nodes are still guaranteed to be colored

Other Extensions

- *Coalescing* merges (non-interfering) copy-related nodes
  - May make graph impossible to color
- *Conservative coalescing* merges only if the merged node is locally colorable when all locally colorable neighbors are removed
- A *spill metric* determines which node to spill. Adapted one takes register classes into account (see paper).

UPPSALA UNIVERSITET

Implementation

- Prototype implemented in IAR Systems C/C++ compiler
- Target: Thumb (of ARM/Thumb)
- Includes extensions from paper
- Hard to compare against other allocators, since framework matters
- Theoretically equivalent to Chaitin and Briggs for applicable targets
  - Regular, with “pre-colored” registers
  - Aligned register pairs

UPPSALA UNIVERSITET

Conclusions

- You can use fast, retargetable, global, graph-coloring register allocation for irregular architectures

- The algorithm degenerates to the standard algorithm for regular architectures, so you can use it for all targets

- Need to improve the quality? Add some more well-proven extensions