Global Register Allocation
via Graph Coloring

Copyright 2003, Keith D. Cooper, Ken Kennedy & Linda Torczon, all rights reserved.
Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.
Register Allocation

Part of the compiler’s back end

Critical properties

• Produce correct code that uses \( k \) (or fewer) registers
• Minimize added loads and stores
• Minimize space used to hold spilled values
• Operate efficiently
  \( O(n), O(n \log_2 n), \) maybe \( O(n^2) \), but not \( O(2^n) \)
Global Register Allocation

The big picture

At each point in the code
1. Determine which values will reside in registers
2. Select a register for each such value
The goal is an allocation that “minimizes” running time

Most modern, global allocators use a graph-coloring paradigm
- Build a “conflict graph” or “interference graph”
- Find a $k$-coloring for the graph, or change the code to a nearby problem that it can $k$-color

Optimal global allocation is NP-Complete, under almost any assumptions.
Global Register Allocation

What’s harder across multiple blocks?

• Could replace a load with a move
• Good assignment would obviate the move
• Must build a control-flow graph to understand inter-block flow
• Can spend an inordinate amount of time adjusting the allocation

This is an assignment problem, not an allocation problem!
Global Register Allocation

A more complex scenario
- Block with multiple predecessors in the control-flow graph
- Must get the “right” values in the “right” registers in each predecessor
- In a loop, a block can be its own predecessors

This adds tremendous complications
Global Register Allocation

Taking a global approach

• Abandon the distinction between local & global
• Make systematic use of registers or memory
• Adopt a general scheme to approximate a good allocation

Graph coloring paradigm \((\text{Lavrov} \ \& \ \text{(later) Chaitin})\)

1. Build an interference graph \(G_I\) for the procedure
   \(\rightarrow\) Computing LIVE is harder than in the local case
   \(\rightarrow\) \(G_I\) is not an interval graph

2. (try to) construct a \(k\)-coloring
   \(\rightarrow\) Minimal coloring is NP-Complete
   \(\rightarrow\) Spill placement becomes a critical issue

3. Map colors onto physical registers
Graph Coloring (A Background Digression)

The problem

A graph $G$ is said to be $k$-colorable iff the nodes can be labeled with integers $1\ldots k$ so that no edge in $G$ connects two nodes with the same label.

Examples

Each color can be mapped to a distinct physical register.
Building the Interference Graph

What is an “interference”? (or conflict)

- Two values *interfere* if there exists an operation where both are simultaneously live
- If $x$ and $y$ interfere, they cannot occupy the same register

To compute interferences, we must know where values are “live”

The interference graph, $G_I$

- Nodes in $G_I$ represent values, or live ranges
- Edges in $G_I$ represent individual interferences
  - For $x, y \in G_I$, $\langle x, y \rangle \in$ iff $x$ and $y$ interfere
- A $k$-coloring of $G_I$ can be mapped into an allocation to $k$ registers
Building the Interference Graph

To build the interference graph

1. Discover live ranges
   - Build SSA form
   - At each $\phi$-function, take the union of the arguments

2. Compute LIVE sets for each block
   - Use an iterative data-flow solver
   - Solve equations for LIVE over domain of live range names

3. Iterate over each block
   - Track the current LIVE set
   - At each operation, add appropriate edges & update LIVE
     - Edge from result to each value in LIVE
     - Remove result from LIVE
     - Edge from each operand to each value in LIVE
What is a Live Range?

- A set LR of definitions \( \{d_1, d_2, \ldots, d_n\} \) such that for any two definitions \( d_i \) and \( d_j \) in LR, there exists some use \( u \) that is reached by both \( d_i \) and \( d_j \).

- How can we compute live ranges?
  - For each basic block \( b \) in the program, compute \( \text{REACHESOUT}(b) \) — the set of definitions that reach the exit of basic block \( b \)
    - \( d \in \text{REACHESOUT}(b) \) if there is no other definition on some path from \( d \) to the end of block \( b \)
  - For each basic block \( b \), compute \( \text{LIVEIN}(b) \) — the set of variables that are live on entry to \( b \)
    - \( v \in \text{LIVEIN}(b) \) if there is a path from the entry of \( b \) to a use of \( v \) that contains no definition of \( v \)
  - At each join point in the CFG, for each live variable \( v \), merge the live ranges associated with definitions in \( \text{REACHESOUT}(p) \), for all predecessors of \( b \), that assign a value to \( v \).
Computing LIVE Sets

A value $v$ is live at $p$ if $\exists$ a path from $p$ to some use of $v$ along which $v$ is not re-defined

Data-flow problems are expressed as simultaneous equations

\[
\text{LIVEOUT}(b) = \bigcup_{s \in \text{succ}(b)} \text{LIVEIN}(s)
\]
\[
\text{LIVEIN}(b) = (\text{LIVEOUT}(b) \cap \text{VARKILL}(b)) \cup \text{UEVAR}(b)
\]

where

- $\text{UEVAR}(b)$ is the set of upward-exposed variables in $b$
  (names used before redefinition in block $b$)
- $\text{VARKILL}(b)$ is the set of variable names redefined in $b$

As output,

- $\text{LIVEOUT}(x)$ is the set of names live on exit from block $x$
- $\text{LIVEIN}(x)$ is the set of names live on entry to block $x$

*solve it with the iterative algorithm*
Observation on Coloring for Register Allocation

- Suppose you have $k$ registers—look for a $k$ coloring

- Any vertex $n$ that has fewer than $k$ neighbors in the interference graph ($n^\circ < k$) can always be colored!
  - Pick any color not used by its neighbors — there must be one

- Ideas behind Chaitin’s algorithm:
  - Pick any vertex $n$ such that $n^\circ < k$ and put it on the stack
  - Remove that vertex and all edges incident from the interference graph
    - This may make some new nodes have fewer than $k$ neighbors
  - At the end, if some vertex $n$ still has $k$ or more neighbors, then spill the live range associated with $n$
  - Otherwise successively pop vertices off the stack and color them in the lowest color not used by some neighbor
Chaitin's Algorithm

1. While \( \exists \) vertices with < \( k \) neighbors in \( G_I \)
   > Pick any vertex \( n \) such that \( n^\circ < k \) and put it on the stack
   > Remove that vertex and all edges incident to it from \( G_I \)
     • This will lower the degree of \( n \)'s neighbors

2. If \( G_I \) is non-empty (all vertices have \( k \) or more neighbors) then:
   > Pick a vertex \( n \) (using some heuristic) and spill the live range associated with \( n \)
   > Remove vertex \( n \) from \( G_I \), along with all edges incident to it and put it on the stack
   > If this causes some vertex in \( G_I \) to have fewer than \( k \) neighbors, then go to step 1; otherwise, repeat step 2

3. Successively pop vertices off the stack and color them in the lowest color not used by some neighbor
Chaitin's Algorithm in Practice

3 Registers

Stack
Chaitin’s Algorithm in Practice

3 Registers

Stack

\[ \begin{align*}
1 & \quad 2 \\
3 & \quad 4 \\
5 &
\end{align*} \]
Chaitin's Algorithm in Practice

3 Registers

Stack

2
1

3
4
5
Chaitin’s Algorithm in Practice

3 Registers

Stack

4
2
1

3
5
Chaitin’s Algorithm in Practice

3 Registers

Stack

5
3
4
2
1

Colors:

1: 
2: 
3: 

Chaitin’s Algorithm in Practice

3 Registers

Stack

Colors:
1: 
2: 
3: 

5
Chaitin’s Algorithm in Practice

3 Registers

Stack

Colors:
1: 
2: 
3: 

Graph:

Nodes: 3, 4, 2, 1, 5
Edges: 3-4, 4-5
Chaitin’s Algorithm in Practice

3 Registers

Stack

Colors:
1:
2:
3:
Chaitin's Algorithm in Practice

3 Registers

Stack

Colors:
1: 
2: 
3: 

Diagram:

- Register 1
- Register 2
- Register 3
- Register 4
- Register 5

Connections:
- Register 1 to Register 2
- Register 1 to Register 3
- Register 2 to Register 4
- Register 3 to Register 4
- Register 4 to Register 5
- Register 5 to Register 4

Colors:
- 1: Yellow
- 2: Pink
- 3: Blue
Chaitin's Algorithm in Practice

3 Registers

Stack

Colors:
1:  
2:  
3:  

Diagram:

1 -- 2
1 -- 3
2 -- 4
3 -- 4
4 -- 5
Improvement in Coloring Scheme

Optimistic Coloring  (*Briggs, Cooper, Kennedy, and Torczon*)

• Instead of stopping at the end when all vertices have at least k neighbors, put each on the stack according to some priority

  → When you pop them off they may still color!

2 Registers:

![Diagram of four vertices connected in a square with arrows indicating the flow of vertices on the stack.](image-url)
Improvement in Coloring Scheme

Optimistic Coloring  \textit{(Briggs, Cooper, Kennedy, and Torczon)}

- Instead of stopping at the end when all vertices have at least $k$ neighbors, put each on the stack according to some priority

  $\rightarrow$ When you pop them off they may still color!

2 Registers:

\begin{center}
\begin{tikzpicture}
  \node[draw, circle, fill=red] at (0,0) (A) {};
  \node[draw, circle, fill=blue] at (1,1) (B) {};
  \node[draw, circle, fill=red] at (1,-1) (C) {};
  \node[draw, circle, fill=blue] at (-1,-1) (D) {};
  \path (A) edge (B);
  \path (A) edge (C);
  \path (A) edge (D);
  \path (B) edge (C);
  \path (B) edge (D);
  \path (C) edge (D);
\end{tikzpicture}
\end{center}

2-colorable
Chaitin-Briggs Algorithm

1. While ∃ vertices with < k neighbors in $G_I$
   - Pick any vertex $n$ such that $n < k$ and put it on the stack
   - Remove that vertex and all edges incident to it from $G_I$
     - This may create vertices with fewer than $k$ neighbors

2. If $G_I$ is non-empty (all vertices have $k$ or more neighbors) then:
   - Pick a vertex $n$ (using some heuristic condition), push $n$ on the stack and remove $n$ from $G_I$, along with all edges incident to it
   - If this causes some vertex in $G_I$ to have fewer than $k$ neighbors, then go to step 1; otherwise, repeat step 2

3. Successively pop vertices off the stack and color them in the lowest color not used by some neighbor
   - If some vertex cannot be colored, then pick an uncolored vertex to spill, spill it, and restart at step 1
Chaitin Allocator (Bottom-up Coloring)

- **renumber**
- **build**
- **coalesce**
- **spill costs**
- **simplify**
- **select**
- **spill**

**Chaitin’s algorithm**

1. **Build SSA, build live ranges, rename**
2. **Build the interference graph**
3. **Fold unneeded copies**
   \[
   LR_x \rightarrow LR_y, \text{ and } <LR_x, LR_y> \notin G_i \Rightarrow \text{combine } LR_x \& LR_y
   \]
4. **Estimate cost for spilling each live range**
5. **Remove nodes from the graph**
6. **While stack is non-empty**
   - pop \( n \), insert \( n \) into \( G_i \), & try to color it
7. **Spill uncolored definitions & uses**

**while \( N \) is non-empty**
- if \( \exists n \text{ with } n^e < k \) then
  - push \( n \) onto stack
- else pick \( n \) to spill
  - push \( n \) onto stack
  - remove \( n \) from \( G_i \)
Chaitin Allocator (Bottom-up Coloring)

1. **renumber**
   - Build SSA, build live ranges, rename

2. **build**
   - Build the interference graph

3. **coalesce**
   - Fold unneeded copies
   - For each live range intersect, LR_x → LR_y, and <LR_x,LR_y> ∉ G_i ⇒ combine LR_x & LR_y

4. **spill costs**
   - Estimate cost for spilling each live range

5. **simplify**
   - Remove nodes from the graph

6. **select**
   - While stack is non-empty
     - pop n, insert n into G_i, & try to color it
   - Spill uncolored definitions & uses

Chaitin’s algorithm

while N is non-empty
if ∃ n with n°< k then
  push n onto stack
else pick n to spill
  push n onto stack
  remove n from G_i
Chaitin-Briggs Allocator (Bottom-up Coloring)

- renumber
- build
- coalesce
- spill costs
- simplify
- select
- spill

Build SSA, build live ranges, rename

Build the interference graph

Fold unneeded copies

\[ LR_x \to LR_y, \text{ and } < LR_x, LR_y > \notin G_i \Rightarrow \text{combine } LR_x \& LR_y \]

Estimate cost for spilling each live range

Remove nodes from the graph

While stack is non-empty

- pop \( n \), insert \( n \) into \( G_i \), & try to color it

Spill uncolored definitions & uses

Briggs’ algorithm (1989)
Picking a Spill Candidate

When \( \forall n \in G_r, n^\circ \geq k \), simplify must pick a spill candidate.

Chaitin's heuristic

- **Minimize spill cost \div current degree**

- If \( LR_x \) has a negative spill cost, spill it pre-emptively
  - \( \rightarrow \) Cheaper to spill it than to keep it in a register

- If \( LR_x \) has an infinite spill cost, it cannot be spilled
  - \( \rightarrow \) No value dies between its definition & its use
  - \( \rightarrow \) No more than \( k \) definitions since last value died \( (safety\ valve) \)

Spill cost is weighted cost of loads & stores needed to spill \( x \)

Bernstein *et al.* Suggest repeating simplify, select, & spill with several different spill choice heuristics & keeping the best.
Other Improvements to Chaitin-Briggs

Spilling partial live ranges
• Bergner introduced interference region spilling
• Limits spilling to regions of high demand for registers

Splitting live ranges
• Simple idea — break up one or more live ranges
• Allocator can use different registers for distinct subranges
• Allocator can spill subranges independently (use 1 spill location)

Conservative coalescing
• Combining $LR_x \rightarrow LR_y$ to form $LR_{xy}$ may increase register pressure
• Limit coalescing to case where $LR_{xy} \prec k$
• Iterative form tries to coalesce before spilling
Chaitin-Briggs Allocator (Bottom-up Global)

Strengths & weaknesses

↑ Precise interference graph
↑ Strong coalescing mechanism
↑ Handles register assignment well
↑ Runs fairly quickly

↓ Known to overspill in tight cases
↓ Interference graph has no geography
↓ Spills a live range everywhere
↓ Long blocks devolve into spilling by use counts

Is improvement still possible?

• Rising spill costs, aggressive transformations, & long blocks
  ⇒ yes, it is
What about Top-down Coloring?

The Big Picture
- Use high-level priorities to rank live ranges
- Allocate registers for them in priority order
- Use coloring to assign specific registers to live ranges

The Details
- Separate constrained from unconstrained live ranges
  - A live range is constrained if it has \( \geq k \) neighbors in \( G_I \)
- Color constrained live ranges first
- Reserve pool of local registers for spilling (or spill & iterate)
- Chow split live ranges before spilling them
  - Split into block-sized pieces
  - Recombine as long as \( \circ < k \)

Unconstrained must receive a color!

Use spill costs as priority function!
What about Top-down Coloring?

The Big Picture

• Use high-level priorities to rank live ranges
• Allocate registers for them in priority order
• Use coloring to assign specific registers to live ranges

More Details

• Chow used an imprecise interference graph
  \[ \langle x, y \rangle \in G_I \iff x, y \in \text{LiveIN}(b) \text{ for some block } b \]
  \[ \Rightarrow \text{Cannot coalesce live ranges since } x \to y \Rightarrow \langle x, y \rangle \in G_I \]
• Quicker to build imprecise graph
  \[ \Rightarrow \text{Chow’s allocator runs faster on small codes, where demand for} \]
  \[ \text{registers is also likely to be lower } \quad \text{(rationalization)} \]
Tradeoffs in Global Allocator Design

Top-down versus bottom-up
- Top-down uses high-level information
- Bottom-up uses low-level structural information

Spilling
- Reserve registers versus iterative coloring

Precise versus imprecise graph
- Precision allows coalescing
- Imprecision speeds up graph construction

Even JITs use this stuff ...
Regional Approaches to Allocation

Hierarchical Register Allocation (Koblenz & Callahan)
• Analyze control-flow graph to find hierarchy of tiles
• Perform allocation on individual tiles, innermost to outermost
• Use summary of tile to allocate surrounding tile
• Insert compensation code at tile boundaries ($LR_x \rightarrow LR_y$)

<table>
<thead>
<tr>
<th>Strengths</th>
<th>Weaknesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>→ Decisions are largely local</td>
<td>→ Decisions are made on local information</td>
</tr>
<tr>
<td>→ Use specialized methods</td>
<td>→ May insert too many copies</td>
</tr>
<tr>
<td>on individual tiles</td>
<td></td>
</tr>
<tr>
<td>→ Allocator runs in parallel</td>
<td>Still, a promising idea</td>
</tr>
</tbody>
</table>

• Anecdotes suggest it is fairly effective
• Target machine is multi-threaded multiprocessor (Tera MTA)
Regional Approaches to Allocation

Probabilistic Register Allocation (Proebsting & Fischer)

- Attempt to generalize from Best’s algorithm (bottom-up, local)
- Generalizes “furthest next use” to a probability
- Perform an initial local allocation using estimated probabilities
- Follow this with a global phase
  - Compute a merit score for each LR as
    (benefit from x in a register = probability it stays in a register)
  - Allocate registers to LRs in priority order, by merit score, working from inner loops to outer loops
  - Use coloring to perform assignment among allocated LRs

- Little direct experience (either anecdotal or experimental)
- Combines top-down global with bottom-up local
Regional Approaches to Allocation

Register Allocation via Fusion (Lueh, Adl-Tabatabi, Gross)

- Use regional information to drive global allocation
- Partition CFGs into regions & build interference graphs
- Ensure that each region is $k$-colorable
- Merge regions by fusing them along CFG edges
  - Maintain $k$-colorability by splitting along fused edge
  - Fuse in priority order computed during the graph partition
- Assign registers using int. graphs
  - i.e., execution frequency

Strengths

- Flexibility
- Fusion operator splits on low-frequency edges

Weaknesses

- Choice of regions is critical
- Breaks down if region connections have many live values
Extra Slides Start Here
Computing LIVE Sets

The compiler solve the equations with an iterative algorithm

\[
\text{WorkList} \leftarrow \{ \text{all blocks} \}
\]

\[
\text{while ( WorkList } \neq \emptyset )
\]

\[
\text{remove a block } b \text{ from WorkList}
\]

\[
\text{Compute } \text{LIVEOUT}(b)
\]

\[
\text{Compute } \text{LIVEIN}(b)
\]

\[
\text{if } \text{LIVEIN}(b) \text{ changed}
\]

\[
\text{then add } \text{pred} (b) \text{ to WorkList}
\]

Why does this work?

- \( \text{LIVEOUT}, \text{LIVEIN} \subseteq 2^{\text{Names}} \)
- \( \text{UEVAR}, \text{VARKILL} \) are constant for \( b \)
- Equations are monotone
- Finite chains in the lattice
  \[ \Rightarrow \] will reach a fixed point!

Speed of convergence depends on the order in which blocks are “removed” & their sets recomputed

This is the world’s quickest introduction to data-flow analysis!