Big Picture of Code Generation

- **Register allocation**
  - Decides which values will reside in registers
  - Changes the storage mapping
  - Concerns about placement of data & memory operations
    - Single basic block, no spilling, and one register size
    - Whole procedure is NP-Complete
    - Start from virtual registers and map “enough” into $k$
    - With targeting, focus on good priority heuristic
Register Allocation

- Part of the compiler’s back end

Critical properties

- Produce correct code that uses $k$ (or fewer) registers
- Minimize running time
  - Minimize added loads and stores
  - Minimize space used to hold spilled values
- NP complete problem
  - Need heuristic
  - Operate efficiently - $O(n)$, $O(n \log_2 n)$, maybe $O(n^2)$, but not $O(2^n)$
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

- Need to discover relationship between definitions and uses.
  - DU-chain, SSA are good representations
  - SSA has information about defs and uses relationship presented with a single name
  - Construct web
    - DU-chain: maximal union of DU-chains
    - SSA: $\emptyset(...)$ unions sets associated with each parameter
**Web from DU-chains**

- Built from DU chains
- The maximal union of intersecting DU-chains for a variable
- Ex) assigning webs to temporaries
Instead of DU-chains, the SSA form can be used to identify webs in the program.

Each SSA-form variable is the head of a DU-chain.

Union is found at $\phi$-function.
Graph Coloring  (A Background Digression)

- The problem
  - A graph $G$ is said to be $k$-colorable iff the nodes can be colored with $k$ different colors so that no edge in $G$ connects two nodes with the same color.

- Examples

- Each color can be mapped to a distinct physical register.
Global Register Allocation

- **Graph coloring paradigm** *(Lavrov & (later) Chaitin)*
  1. Build an interference graph $G_I$ for the procedure
     - Computing LIVE is harder than in the local case
     - $G_I$ is not an interval graph
  2. Try to construct a $k$-coloring
     - Minimal coloring is NP-Complete
     - Spill placement becomes a critical issue
  3. Map colors onto physical registers
Building the Interference Graph (1)

- **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

- **The interference graph, $G_I$**
  - Nodes in $G_I$ represent values, webs or live ranges
  - Edges in $G_I$ represent individual interferences
    - For $x, y \in G_I$, $<x,y> \in$ iff $x$ and $y$ interfere
  - A $k$-coloring of $G_I$ can be mapped into an allocation to $k$ registers
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 LIVEOUT over domain of live ranges
   - Draw edges among live ranges in LIVEOUT

3. Iterate over each block
   - Track the current LIVENOW set backward from the bottom
   - At each operation, add appropriate edges & update LIVENOW
     ✓ With operation form of $\text{op } r1, r2 \rightarrow r3$
     for each $r$ in LIVENOW draw an edge $(r3, r)$
     remove $r3$ from LIVENOW
     add $r1, r2$ to LIVENOW
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 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 $n$. Then a new code is generated with spill.
   - Build $G_I$ from the new code and restart at step 1.

3. Successively pop vertices off the stack and color them in the lowest color not used by already colored neighbors
Chaitin’s Algorithm in Practice

3 Registers

Stack

Diagram of a graph with nodes 1, 2, 3, 4, and 5 connected in a specific pattern.
Chaitin’s Algorithm in Practice

3 Registers

Stack

1

2

4

3

5
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

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:  

4  
2  
1  

3

5
Chaitin’s Algorithm in Practice

3 Registers

Stack

Colors:
1: 
2: 
3: 

3 4 5

2 1
Chaitin’s Algorithm in Practice

3 Registers

Stack

Colors:
1:  
2:  
3:  

Diagram showing connections between registers and a stack.
Chaitin’s Algorithm in Practice

3 Registers

Stack

Colors:
1:  
2:  
3:  

Diagram showing nodes and edges with colors assigned to them.
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:

2-colorable
Chaitin-Briggs Algorithm

1. **While ∃ vertices with < k neighbors in** $G_I$
   - Pick any vertex $n$ such that $n^o < k$ 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 already colored neighbors**
   - If some vertex cannot be colored, then pick an uncolored vertex to spill, spill it.
   - Build $G_I$ from the new code and restart at step 1.
Chaitin Allocator

(Bottom-up Coloring)

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

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

3. **coalesce**
   - Fold unneeded copies
   - \( LR_x \rightarrow LR_y \), and \(< LR_x, LR_y > \) $\notin G_i \Rightarrow$ combine \( LR_x \) & \( LR_y \)

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

5. **simplify**
   - Remove nodes from the graph and push to the stack

6. **select**
   - While stack is non-empty
   - pop \( n \), insert \( n \) back into \( G_i \), & try to color it

7. **spill**
   - Spill uncolored definitions & uses
**Chaitin-Briggs Allocator (Bottom-up Coloring)**

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

---

**Build SSA, build live ranges, rename**

**Build the interference graph**

**Fold unneeded copies**

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

**Estimate cost for spilling each live range**

**Remove nodes from the graph & push to the stack**

**While stack is non-empty**

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

**Spill uncolored definitions & uses**
Picking a Spill Candidate

- **Spill cost metric**
  - Weighted cost of loads & stores needed to spill x
  - Repeat simplify, select, & spill with several different spill choice heuristics and keep the best (Bernstein *et al*).

- **Chaitin’s heuristic**
  - Minimize spill cost and reduce current degree of $G_I$
  - A few special cases
    - Negative spill cost (spill preemptively)
      - load and store to the same memory location, but no other uses
    - Infinite spill cost (cannot be spilled)
      - A use immediately follows its definition (short Live-Range)
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

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

- **Strengths**
  - Precise interference graph
  - Strong coalescing mechanism
  - Handles register assignment well
  - Runs fairly quickly

- **Weaknesses**
  - Known to overspill in tight cases (overspill in high demand)
  - Interference graph has no geography (size of overlapping range)
  - Spills a live range everywhere (no partial spill)
Summary

- **Register allocation**
  - Infinite register code to finite register code
  - Spill may be needed
  - Graph coloring problem

- **Chaitin-Briggs algorithm**
  - Build interference graph
  - Simplify graph with stack
  - Optimistic spill

- Many more tweaks are possible to improve