**Prelude**
- Who is Kazushige Goto (I don’t expect you to know this one)
- Japanese patent clerk
- Started working on hand-tuning scientific code in his free time
- Mostly linear algebra (e.g., matrix multiply)
- He does the crazy stuff:
  - Unroll a loop some number of times
  - During execution, put the jump in the right place
- Expert in tuning for Pentium
- Worked at U of Texas
  - See recent article in NY Times
- Pentium 4 cluster: 1.5 teraflops to 2.0 teraflops

**Register allocation**
- What are registers?
  - Memory
  - Very close to the processor — very fast to access
  - On many architectures, required by ISA
  - RISC – all computations use registers
  - Pentium – many instructions register + memory
- Part of the memory hierarchy
  - Top: close to CPU, fast, small
  - Bottom: far from CPU, slow, large

**Memory hierarchy**
- What is the compiler’s role in the memory hierarchy?
  - Virtual memory?
  - Main memory?
  - Heap layout
  - Prefetching
  - Level-1 and level-2 cache?
  - Many locality optimizations
  - Loop transforms, tiling, strip mining
  - Registers
  - Compiler has direct control
Using registers

- Machine code: register names are explicit
  - Represent data dependences
  - Renaming may occur inside the processor
  - Alpha ISA: 32 integer, 32 floating point registers
  - Alpha 21264: 80 integer, 72 floating point registers
  - Why have more physical registers than ISA?

- How important is register allocation?
  - Widely recognized as the most important "optimization" performed by the compiler
  - An order of magnitude compared to poor or no register allocation
  - Most other optimizations: at most ~10% to 20%

Register allocation

- What is the problem?
  - Register allocation
    - Decide which values will be kept in registers
  - Register assignment
    - Select specific registers for each use

- Constraints
  - Primary: limited number of registers
  - Different kinds of registers — integer vs floating point
  - Special-purpose registers — SP
  - Instruction requirements — x86 mul must use eax, adx
  - Some values cannot go in registers

Register allocation

- What values can go in registers?
  - First, what does it mean to "allocate a variable in a register"
    - Most cases: variable becomes a register
    - All uses and defs replaced with the register
    - It has no storage on the stack

- What is the implication of that decision?
  - The compiler must be able to see all accesses
  - For example:
    ```
    int x;
    int *p = &x;
    (*p) = 7;
    foo(p);
    ```
    Might be able to handle (*p) = 7 case

Example

- Key idea: If we can color the graph with K colors, then we can allocate the variables to K registers

Example

- Graph is 2-colorable
Static single assignment

- What is the effect of SSA form on liveness?
- What does SSA do?
  - Breaks a single variable into multiple instances
  - Instances represent distinct, non-overlapping uses
- Effect:
  - Breaks up live ranges – often improves register allocation

Graph coloring

The big questions:

- Can we efficiently find a K-coloring of the graph?
- Can we efficiently find the optimal coloring of the graph (i.e., using the least number of colors)?
- What do we do when there aren’t enough colors (registers) to color the graph?

Graph coloring

- The bad news:
  - Graph coloring is NP-complete
- What does the optimal algorithm do?
  - Works on any graph
  - Tells us for certain if a graph is K-colorable
- Observations
  - We’ll never see the worst-case graph
  - We don’t necessarily need the perfect coloring

Spilling

- What if the graph is not K-colorable?
  - There aren’t enough registers to hold all variables
  - This happens a lot
  - Pick a variable, spill it back to the stack
    - Value lives on the stack
    - We have to generate extra code to load and store it
    - Need registers to hold value temporarily
    - Simple approach: keep a few registers around just for this purpose
    - Better approach:
      - Rewrite the code introducing a new temporary
      - Use the temporary to “load” and “store” the spilled variable
      - Rerun the liveness analysis and register allocation

Rewriting the code

Example: \texttt{add v1, v2}

Suppose v2 is selected for spilling and assign to stack location [SP+12]

Add a new variable t23 just for this instruction:

\begin{verbatim}
  mov [SP+12], t23
  add v1, t23
\end{verbatim}

Idea:

- t23 has a short live range and (hopefully) doesn’t interfere with other variables as much as v2
- Rerun the whole algorithm

Graph coloring

- Assume you have K registers – looking for K-coloring
- Observation:
  - Any node with less than K neighbors (degree < K) must be colorable
  - Why?
  - Pick the color not used by any neighbor
  - There must be one!
- This is the basis for Chaitin’s algorithm (Chaitin, 1981)
Chaitin’s algorithm

Ideas behind Chaitin’s algorithm:

- Pick any vertex \( n \) such that \( n^\circ < k \) and put it on the stack
- Remove that vertex from the interference graph
- And incident edges
- Goal: 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

1 2 3

4 Registers

Stack

1 2 3 4

Chaitin’s Algorithm in Practice

3 Registers

Stack

2 1

4 Registers

Stack

4 2 1

5 Registers
Improvements

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 \( \exists \) vertices with \(< k \) neighbors in \( G_I \):
   - Pick any vertex \( n \) such that \( n \leq k \) and put it on the stack
   - Remove that vertex and all edges incident to it from \( G_I \)
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

1. Build SSA, build live ranges, rename
2. Build the interference graph
3. Fold unneeded copies
4. \( LR_x \rightarrow LR_y \) and \( < LR_x,LR_y> \) \( \notin GI \) \( \Rightarrow \) combine \( LR_x \) & \( LR_y \)
5. Remove nodes from the graph
6. While stack is non-empty
   - pop \( n \), insert \( n \) into \( GI \), & try to color it
7. Estimate cost for spilling each live range
8. While \( N \) is non-empty
   - if \( \exists \) \( n \) with \( n \leq k \) then
     - push \( n \) onto stack else pick \( n \) to spill
   - push \( n \) onto stack
   - remove \( n \) from \( G_I \)
9. Spill uncolored definitions & uses

Chaitin-Briggs Allocator

1. Build SSA, build live ranges, rename
2. Build the interference graph
3. Fold unneeded copies
4. \( LR_x \rightarrow LR_y \) and \( < LR_x,LR_y> \) \( \notin GI \) \( \Rightarrow \) combine \( LR_x \) & \( LR_y \)
5. Remove nodes from the graph
6. While stack is non-empty
   - pop \( n \), insert \( n \) into \( GI \), & try to color it
7. Estimate cost for spilling each live range
8. While \( N \) is non-empty
   - if \( \exists \) \( n \) with \( n \leq k \) then
     - push \( n \) onto stack else pick \( n \) to spill
   - push \( n \) onto stack
   - remove \( n \) from \( G_I \)
9. Spill uncolored definitions & uses
Picking a spill candidate

- Critical heuristic – spilling can be expensive
- Goal: minimize the performance impact
  - Spilled variable must be stored at each def, loaded at each use
  - Higher degree nodes interfere with more variables
  - Chaitin: minimize \( \text{spill.cost} + \text{current degree} \)
- Many subtle variations
  - Live range splitting
  - More sophisticated spill cost estimation
  - Impact on rest of the coloring problem
  - Interaction with other optimizations – scheduling, copy propagation

A different approach

- What if graph coloring approach is still too expensive?
  - Example: in a just-in-time compiler
    - Compilation time is critical
    - Compiler needs to be simple and fast
  - Interference graph has worst-case quadratic size
- Alternative: Linear scan register allocation (Poletto, 1999)
  - Make one pass over the list of variables
  - Spill variables with longest lifetimes – those that would tie up a register for the longest time

Linear scan

- First: Compute live intervals
  - Linearize the IR – usually just a list of tuples/instructions
  - A live interval for a variable is a range \([i, j]\)
    - The variable is not live before instruction \(i\)
    - The variable is not live after instruction \(j\)
- Idea: overlapping live intervals imply interference
  - Given \(R\) registers and \(N\) overlapping intervals
    - \(R\) intervals allocated to registers
    - \(N-R\) intervals spilled to the stack
  - What does this imply about the linearization?

Algorithm

- Sort live intervals
  - In order of increasing start points
  - Quickly find the next new interval
- Maintain a sorted list of active intervals
  - In order of increasing end points
  - Quickly find expired intervals
- At each step, update active as follows
  - Add the next interval from the sorted list
  - Remove any expired intervals (those whose end points are earlier than the start point of the new interval)

Example

- Variables
- Algorithm
  - Extra restriction: Never allow active to have more than \(R\) elements
  - Spill scenario: active has \(R\) elements, new interval doesn’t cause any existing intervals to expire
  - Heuristic: Spill the interval that ends last (furthest away from the current position)
    - Has optimal behavior for straight-line code
    - Appears to work well even in linearized code

Example (2 registers)

- Step 1: active = \([a]\)
- Step 2: active = \([a, b]\)
- Step 3: active = \([a, b, c]\) → spill \(c\) → active = \([a, b]\)
- Step 4: \(a\) and \(b\) expire, active = \([d]\)
- Step 5: active = \([d, e]\)
Linear scan

- Register allocation
  - Each new interval added to active gets the next register
  - Registers freed as intervals are removed

- Resulting code: within 10% of graph coloring

- Compilation time: 2 – 3 times faster than graph coloring

- Architectural considerations
  - How sensitive is architecture to register allocation?
  - Many registers (Alpha, PowerPC): use linear scan
  - Few registers (x86): use graph coloring

Next time

- Only two more classes

- Possible topics
  - More on optimizations
  - Memory management
  - Linking and loading
  - Instruction scheduling
  - Compiling other (non-imperative languages)