Prelude

- What memory technology is this?
  - Magnetic core
- My computer has 2GB of RAM, what kind of memory is that?
  - DRAM – dynamic RAM
  - What’s nice about DRAM?
    - 1 transistor, 1 capacitor – very high density
  - What’s bad about DRAM?
    - Capacitors leak – must be refreshed
- What other technologies are there?
  - SRAM – static RAM
  - Flash – NAND and NOR gates
  - MRAM – magnetic RAM

Dimensions:
- Density
- Power
- Speed
- Degradation

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

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
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:
    ```c
    int x;
    int * p = &x;
    (*p) = 7;
    foo(p);
    ```
    - Might be able to handle (*p) = 7 case

Example
- Primary problems to be solved:
  - Usually more variables than registers
  - Can’t use the same register for two variables that are live at the same time
- Key insight:
  - We can cast this as a graph coloring problem
    - Nodes = program variables
    - Edges = connect variables that are live at the same time
    - “Interference graph” or “conflict graph”
    - Colors represent registers

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
  ```c
  R2 = R1 + 2
  R2 = R2 * R2
  R2 = R2 + 1
  return R2*R1
  ```
<table>
<thead>
<tr>
<th>Code</th>
<th>Live sets</th>
<th>Live ranges</th>
<th>Interference graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>b = a + 2</td>
<td>(a,b)</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>c = b * b</td>
<td>(a,c)</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>b = c + 1</td>
<td>(a,b)</td>
<td>return b*a</td>
<td></td>
</tr>
</tbody>
</table>

= Register 1 (R1)
= Register 2 (R2)
Scope

- Simple formulation:
  - Within a basic block – called **local**
  - Live ranges are linear – just look at how they overlap
  - At basic block boundaries:
    - Load into registers on entry
    - Store to memory on exit

- More sophisticated:
  - Across the control-flow graph – called **global**
  - Consider live ranges as “webs” of dependences
  - **Key**: use the same graph coloring algorithm

Example

```
def y
  def x
  use y
  use x
  use x
```

Example

```
def y
  def x
  use y
  use x
  use x
```

Example

```
def y
  def x
  use y
  use x
  use x
```

Example

```
def y
  def x
  use y
  use x
  use x
```

Example

```
def y
  def x
  use y
  use x
  use x
```

Example

```
def y
  def x
  use y
  use x
  use x
```
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
- Compute an approximation with heuristics
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: \( \text{add } v1, v2 \)
- Suppose \( v2 \) is selected for spilling and assign to stack location \([SP+12]\)
- Add a new variable \( t23 \) just for this instruction:
  \[
  \begin{align*}
  \text{mov } & [SP+12], t23 \\
  \text{add } & v1, t23 \\
  \end{align*}
  \]
- Idea: \( t23 \) has a short live range and (hopefully) doesn’t interfere with other variables as much as \( v2 \)
- Rerun the whole algorithm

More spilling
- Problem:
  - This approach turns a single large live range into many small live ranges with many loads and stores
  - Can we do better?
- Live range splitting
  - Choose a point in the live range – insert a store followed by a load
  - Divides the live range into two (or more pieces)
  - Key: choose carefully to reduce the degree of nodes

Another improvement
- Register coalescing
  - We may be able to reduce the degree of vertices by merging live ranges that are connected only by a copy
  - Idea:
    - Find a register copy “\( tb = ta \)”
    - If \( t1 \) and \( t2 \) do not interfere, combine their live ranges
  - Example:
    \[
    \begin{align*}
    \text{add } t1, t2, ta \\
    \cdots \\
    \text{mov } & ta, tb \\
    \text{mov } & ta, tc \\
    \cdots \\
    \text{add } & tb, t3, t4 \\
    \text{add } & tc, t5, t6 \\
    \end{align*}
    \]

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

Chaitin’s algorithm
- Idea:
  - Pick any vertex \( n \) with fewer than \( k \) neighbors
    - This is a k-colorable vertex
  - Remove that vertex from the interference graph
    - Also: remove incident edges
    - Key: this may result in some other nodes now having fewer than \( k \) neighbors
  - If we get stuck, spill the variable whose node has more that \( k \) neighbors, and continue
Chaitin’s Algorithm

1. While ∃ vertices with < k neighbors in G_k:
   - 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_k
   - This will lower the degree of n’s neighbors

2. If G_k 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_k, along with all edges incident to it and put it on the stack
   - If this causes some vertex in G_k 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

Colors:
1: 🔵
2: 🔷
3: 🔵

Stack

Chaitin’s Algorithm in Practice

3 Registers

Colors:
1: 🔵
2: 🔷
3: 🔵

Stack

Chaitin’s Algorithm in Practice

3 Registers

Colors:
1: 🔵
2: 🔷
3: 🔵

Stack

Chaitin’s Algorithm in Practice

3 Registers

Colors:
1: 🔵
2: 🔷
3: 🔵

Stack

Chaitin’s Algorithm in Practice

3 Registers

Colors:
1: 🔵
2: 🔷
3: 🔵

Stack

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:
**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:

![Graph](image)

2-colorable

---

**Chaitin-Briggs 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 $G_I$

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 $LR_x \rightarrow LR_y$ and $< LR_x, LR_y> \notin GI$ ⇒ 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

---

**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 spill cost \( \div \) 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
Allocation constraints

- How do we deal with architectural constraints?
  - Register types (floating point versus integer)
  - Reserved registers – the stack pointer
  - Instruction-level constraints
  - Instruction requirements – x86 mul must use eax, edx
- We can encode constraints in the graph
  - Precolored nodes (for required registers)
  - Additional nodes and edges for constraints
  - Example: explicit nodes for physical registers

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
  - 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?
- Key: choosing the right intervals to spill

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)
- 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

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

Next time
- More on optimization
- A new homework
- Hopefully, new programming assignment