



COMP 181


Lecture 19 Register allocation

November 30, 2005

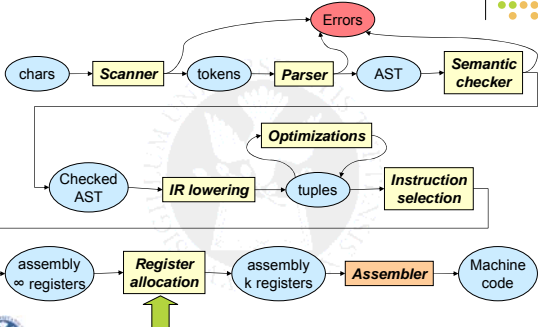
Prelude

- 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



2


Where are we



3

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




4

Memory hierarchy

Farther away, larger, slower

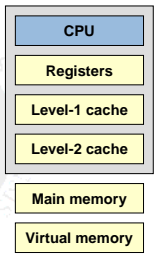

	Pentium 4 3.2 Ghz	Athlon XP 3200	Athlon 64
CPU			
Registers	1 cycle	1 cycle	1 cycle
Level-1 cache	2 cycles (16 KB*)	3 cycles (128 KB)	3 cycles (128 KB)
Level-2 cache	19 cycles (2 MB)	20 cycles (512 KB)	13 cycles (1 MB)
Main memory	204 cycles	180 cycles	125 cycles
Virtual memory	millions of cycles	millions of cycles	millions of cycles



5

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

6

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

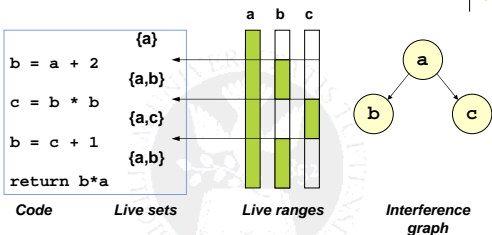


Register allocation

- 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 (Lavrov, Chaitin)
 - Nodes = program variables
 - Edges = connect variables that are live at the same time
 - "Interference graph" or "conflict graph"
 - Colors represent registers



Example

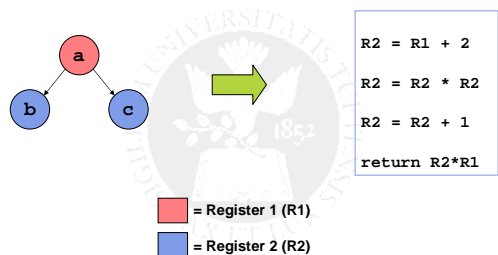


- Key idea:** if we can color the graph with K colors, then we can allocate the variables to K registers



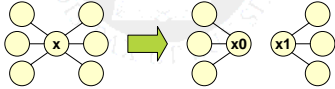
Example

- 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

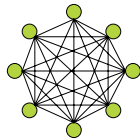
The big questions:

- 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
- 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: `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:


```
mov [SP+12], t23
add v1, t23
```
- 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



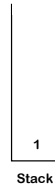
Chaitin's Algorithm in Practice

3 Registers



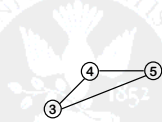
Chaitin's Algorithm in Practice

3 Registers



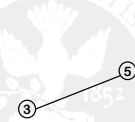
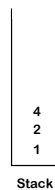
Chaitin's Algorithm in Practice

3 Registers



Chaitin's Algorithm in Practice


3 Registers



Chaitin's Algorithm in Practice

3 Registers

5
3
4
2
1
Stack



Colors:

1: ●

2: ●


3: ●

Tufts University Computer Science
25

Chaitin's Algorithm in Practice

3 Registers

3
4
2
1
Stack



Colors:

1: ●

2: ●

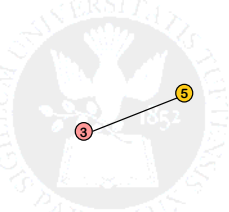
3: ●

Tufts University Computer Science
26

Chaitin's Algorithm in Practice

3 Registers

4
2
1
Stack



Colors:

1: ●

2: ●

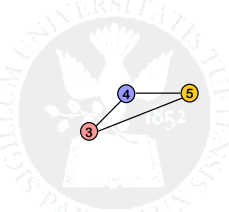
3: ●

Tufts University Computer Science
27

Chaitin's Algorithm in Practice

3 Registers

2
1
Stack



Colors:

1: ●

2: ●

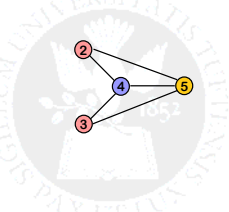
3: ●

Tufts University Computer Science
28

Chaitin's Algorithm in Practice

3 Registers

1
Stack



Colors:

1: ●

2: ●

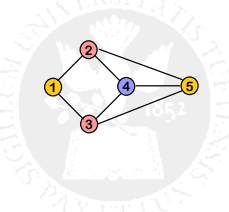
3: ●

Tufts University Computer Science
29

Chaitin's Algorithm in Practice

3 Registers

Stack



Colors:

1: ●

2: ●

3: ●

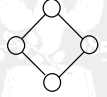
Tufts University Computer Science
30

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:

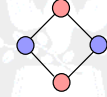


Improvements

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:

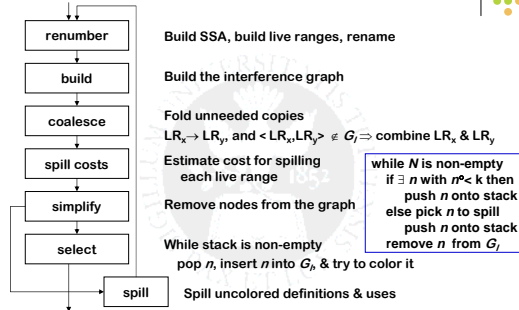


Chaitin-Briggs Algorithm

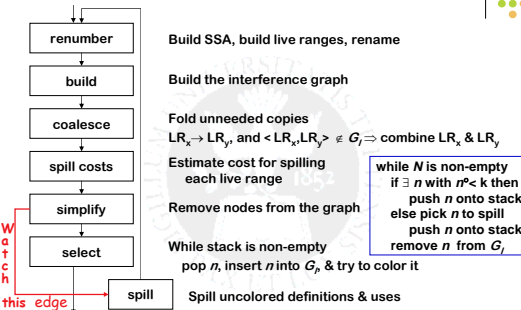
1. While \exists vertices with $< k$ neighbors in G_i
 - > Pick any vertex n such that $n^d < k$ and put it on the stack
 - > Remove that vertex and all edges incident to it from G_i
2. If G_i is non-empty (all vertices have k or more neighbors) then:
 - > Pick a vertex n (using some heuristic condition), 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



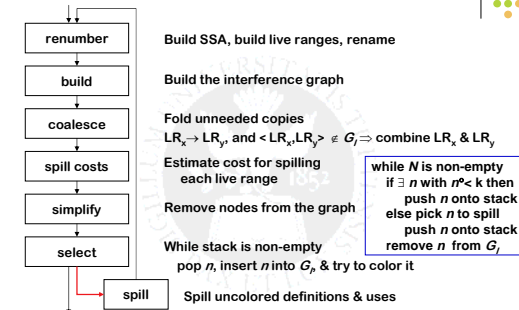
Chaitin Allocator



Chaitin Allocator



Chaitin-Briggs Allocator



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?
- **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*)

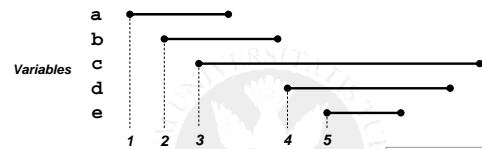


Algorithm

- 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: $\text{active} = \{a\}$
- Step 2: $\text{active} = \{a, b\}$
- Step 3: $\text{active} = \{a, b, c\} \rightarrow$ spill $c \rightarrow \text{active} = \{a, b\}$
- Step 4: a and b expire, $\text{active} = \{d\}$
- Step 5: $\text{active} = \{d, e\}$



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)

