Code generation for modern processors

What are the dominant performance issues for a superscalar RISC processor?

Strategy

select \rightarrow schedule \rightarrow allocate \rightarrow schedule

- select is fairly simple (problem of the 80's)
- allocate and schedule are complex

Definitions (1 of 2)

Instruction selection
- the process of mapping IL into assembly code
- assumes a fixed storage mapping (code shape)
- combining instructions, using address modes

Register allocation
- the process of deciding which values reside in registers
- changes storage mapping (and the code)
- concern about placement of data

Definitions (2 of 2)

Instruction scheduling
- the process of reordering instructions to hide latencies
- assumes a fixed program
- changes demand for registers

Register allocation

<table>
<thead>
<tr>
<th>Concept:</th>
</tr>
</thead>
<tbody>
<tr>
<td>m register allocator \rightarrow \text{register} \rightarrow k register</td>
</tr>
</tbody>
</table>

Assumptions
- Load-store RISC architecture
- Three-address IL
- Previous analysis identifies values that are illegal to hold in registers. Which?
- Load into register before use
- Store back after def

Goals
- Produce correct k register code
- Minimize added loads and stores
- Minimize memory space needed to hold spills
- Allocator must be efficient (no backtracking)
Register Allocation Preliminaries

Definitions and observations

1. Spill virtual register (aka value) \( V \) =
   - Assign to a memory location; not a physical register
   - Load just before each use
   - Store just after each def
   - Usually assign to a stack slot
   - Some algorithms may spill a value in one interval and allocate to a
     register in another

2. Reserve registers to ensure feasibility default
   - (a) must be able to compute addresses, load, & store
   - (b) requires a minimal number of registers, \( F \) : feasible set
   - (c) \( F \) depends on architecture
   - \( \text{MAXLIVE} = \) Maximum number of values live at any instruction (in some region of code, e.g., a basic block)

Allocation versus assignment

The distinction:
- allocation : choosing what to keep in registers at each point
- assignment : choosing specific registers for values

Complexity

<table>
<thead>
<tr>
<th></th>
<th>Allocation</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>optimal, linear time methods for simplest case</td>
<td>uniform regs, no spilling ( \Rightarrow ) linear time</td>
</tr>
<tr>
<td></td>
<td>almost everything else is ( \text{NP}-\text{complete} )</td>
<td>adjacent register pairs ( \Rightarrow ) ( \text{NP}-\text{complete} )</td>
</tr>
<tr>
<td>Global</td>
<td>( \text{NP}-\text{complete} ) for 1 register machine</td>
<td>( \text{NP}-\text{complete} ) for ( k ) register machine</td>
</tr>
<tr>
<td></td>
<td>most subproblems are ( \text{NP}-\text{complete} )</td>
<td>( \text{NP}-\text{complete} )</td>
</tr>
</tbody>
</table>
## Two Approaches for Single Basic Block

**(2) Bottom-up allocation:** Linear scan using live ranges

- A register can hold different values at different statements
- Keep a stack of free registers
- For each statement \( v_3 = v_1 \text{ op } v_2 \)
  - Assign free registers to \( v_1, v_2, v_3 \) (if not in register already)
  - Free a register at the end of a live range
- If no register available, spill some busy register

**Key question:** Which register to spill?

- Spill register used farthest in the future (Sheldon Best 1955)
- On a tie, favor value that need not be stored back to memory

## Global Register Allocation: An Early Approach

**Global extension of Usage Counts**

- Extend usage counts to account for loops, branches
- Insert load at block entry; store at block exit (live values only)
- Some cross-block analysis:
  - Try to avoid load, store at block boundaries

*Few extra spills in the wrong places can be extremely expensive*

## Global Register Allocation: The Modern Approach

**A fundamentally global approach:** Graph Coloring

- Abandon the distinction between local and global
- Model live ranges of entire procedure in a single graph
- Reduce allocation problem to coloring nodes in the graph
- Minimal coloring is \( \text{NP} \)-Complete
  - Use heuristics to choose coloring
- Map colors onto physical registers

## Graph Coloring

**The problem**

A graph \( G = (N, E) \) is said to be \( k \)-colorable if and only if the nodes can be labeled with integers \( 1, \ldots, k \) so that no edge in \( G \) connects nodes with the same label.

**Examples**

- "Diamond" graph 2-colorable
- A more complex graph 3-colorable

**Application to allocation & assignment**

- Graphical representation of conflicts
- Coloring corresponds to feasible assignment
- Model machine constraints in graph
Graph Coloring Register Allocation

4 major aspects:
1. Constructing global live ranges
   - not the same as intuitive live range in straight-line code
2. Building interference graph for a procedure
   - captures information about overlapping live ranges
3. Estimating spill costs
   - important to consider when k-coloring fails
4. (Try to) construct a k-coloring
   - if unsuccessful, choose values to spill and repeat
   - spill placement becomes critical issue

Let's take these one by one.

Global Live Ranges

Definition: Live Range

Also called a web
A live range is a set of references (definitions and uses) s.t.
- for any use in the set, all defs that reach it are in the set too
- for any def in the set, all uses it reaches are in the set too

Fundamental Invariant
- All references in a live range are allocated to the same physical register.

Information needed to compute live ranges:
- need all definitions that reach a single use, and vice versa
- SSA form provides exactly this information:
  - each SSA variable has a single def and zero or more uses
  - \( \phi \)-functions show multiple defs that reach a use:
    \[ x_3 \leftarrow \phi(x_1, \ldots, x_n) \]

The concept of interference

Definition of Interference

- Idea: Two values cannot be in one register if "used in overlapping intervals"
- One way to define "interferes": \( n_i \) and \( n_j \) interfere if they are simultaneously live
  - Problem: What if both are not used in some basic block (where both are live)?
  - Better way: \( n_i \) and \( n_j \) interfere if \( n_i \) is defined at a point where \( n_j \) is live (or vice versa)

Representing the interference property

- model the problem with an interference graph, \( I \)
  - nodes represent values
  - any two values that interfere are connected by an edge
  - a k-coloring for \( i \) fits in \( k \) registers

Computing Global Live Ranges

Idea:
Partition the SSA references into disjoint sets:
- all references to a variable belong in the same set
- all arguments to a \( \phi \)-function belong in the same set as the output variable of the function

Algorithm:
Use the disjoint set union-find algorithm:
1. initially: every name is a separate set
2. repeatedly merge sets that meet at a \( \phi \)-function:
   - \( X = \phi(Y, Z) \): Merge the sets currently holding \( Y \) and \( Z \) into the set holding \( X \)
3. Finally, treat all references in the same live range as a single virtual register
Building the interference graph

Algorithm:
1. Identify global live ranges
2. Build \texttt{LIVE_IN}, \texttt{LIVE_OUT} sets for each block
3. Walk backwards through each block separately:
   (a) Initialize live set: \texttt{LiveNow} $\leftarrow \texttt{LIVE_OUT}$
   (b) For each instruction:
      \( v_1 \ op \ v_2 \rightarrow v_3 \) (i) \( v_3 \) interferes with every value in \texttt{LiveNow}
      (ii) remove \( v_3 \) from \texttt{LiveNow}
      (ii) add \( v_1, v_2 \) to \texttt{LiveNow}

Use two representations:
1. adjacency matrix: lower-diagonal bit matrix
   allows test for interference in \( O(1) \) time
   hash-table has same benefit with lower memory usage for large graphs
2. adjacency list:
   allows efficient iteration over neighbors of a node

Copy Coalescing

An important optimization folded into register allocation

When to coalesce
\[
\text{mov } v_i \rightarrow v_j
\]
Coalesce if:
1. \( v_i \) and \( v_j \) do not interfere, on
2. \( v_i \) and \( v_j \) are not modified after the copy
   i.e., values remain equal always

Coalesce means:
1. Replace \( v_j \) with \( v_i \)
2. Remove copy instruction
3. Combine nodes in the interference graph

Estimating spill costs

Components of cost (per reference) for a spill:

\textbf{Load / store}:
1. address computation
2. memory operation
3. estimated execution frequency
\textbf{Rematerialization}:
1. recomputing the value
2. estimated execution frequency

Address computation:
\( \Rightarrow \) minimize by keeping spilled values in activation record
\( \Rightarrow \) can load / store with (\( p + \) offset) address

Execution frequencies:
\( \Rightarrow \) static estimation:
   \( \Rightarrow \) weight by \( 10^d \) for loop depth \( d \)
   \( \Rightarrow \) weight by branching probability if enclosed in branches
\( \Rightarrow \) Profiling:
   \( \Rightarrow \) measure execution frequencies for representative inputs

Coloring by Graph Pruning: Observations

Two Key Observations
1. The degree < \( k \) rule:
   A graph having a node \( n \) with degree < \( k \) is \( k \)-colorable
   iff the graph with node \( n \) removed is \( k \)-colorable
   \textbf{Proof}:
   \( \Rightarrow \) obvious
   \( \leftarrow \) given \( k \)-coloring of graph without node \( n \), all neighbors of \( N \) use fewer than \( k \) colors. Pick a remaining color for \( n \).
2. Consider a node \( n \) with degree \( \geq k \):
   \( \Rightarrow \) Neighbors of \( N \) may still use fewer than \( k \) distinct colors
   \( \Rightarrow \) defer spilling until you are sure it is needed

\textbf{Idea: Simplify graph by removing nodes}
Repeat until graph is empty:
1. Repeatedly remove a node with degree < \( k \) from the graph
2. When no such node exists: choose a candidate to spill and remove it
### Coloring by Graph Pruning: Algorithm

**Algorithm**

```
Use stack of nodes

while N is non-empty
    if \exists node n with n° < k, push n on stack
    else, pick n as possible spill candidate and push n on stack
    remove n from I (along with its incident edges)

while stack is non-empty
    pop n, insert n into I, try to color n
    if fail to color n, mark n for spilling

Spill all marked nodes (insert code)
```

**Heuristic for choosing spill candidates is key**

### Observations about Reserving Registers

**Observations**

- If reserved registers for executing spill code:
  - after spilling all nodes, we are done
- If did not reserve registers:
  - use virtual registers for spilling
  - repeat entire allocation algorithm
  - more expensive but could give fewer spills

### Chaitin-Briggs register allocators

**Incorporates deferred spilling (Briggs 1989)**

- find live ranges and rename them
- build the interference graph, \( I = (N, E) \)
- find unneeded copies: \( l_i \rightarrow l_j \land \{l_i, l_j\} \notin E \rightarrow \) combine \( l_i \) & \( l_j \)
- estimate cost for spilling each live range
- while \( N \) is non-empty
  - if \( \exists n \) with \( n° < k \), push \( n \) on stack
  - else, put \( n \) for possible spilling & push it
    - remove \( n \) from \( I \)
- while stack is non-empty
  - pop \( n \), insert \( n \) into \( I \), & try to color it

### Picking a spill candidate

**When \( \forall n \in N, n° \geq k \), it must pick a spill candidate**

**Chaitin’s heuristic**

Chaitin says “minimize spill cost” where

- current degree is the number of remaining neighbors
- spill cost of \( l_i \) is defined as

\[
\sum_{j \in \text{refs}(l_i)} 10^{d(j)} - \sum_{j \in \text{loade}(l_i)} 2 \cdot 10^{d(j)} - \sum_{j \in \text{store}(l_i)} 2 \cdot 10^{d(j)}
\]

where

- \( \text{refs}(l_i) \) is \( \text{defs}(l_i) \cup \text{uses}(l_i) \)
- \( d(j) \) is the nesting depth of \( j \)

Bernstein et al. suggested repeating simplify, select, & spill with different spill choice heuristics.
# Improvements to Chaitin: Best of 3 spilling

## The idea
- When allocator blocks, it chooses a value to spill
- Determines which value spills & where code is inserted
- Spill choice is the critical issue

Let

$$area(l) = \sum_{i \in l} num\_live(i) \times 5^{d(i)}$$

## Author

<table>
<thead>
<tr>
<th>Author</th>
<th>Ratio to minimize</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chaitin</td>
<td>Cost current degree</td>
</tr>
<tr>
<td>Bernstein</td>
<td>Cost current degree</td>
</tr>
<tr>
<td>Bernstein</td>
<td>Cost cost</td>
</tr>
</tbody>
</table>

## The implementation
- No metric dominates others (NP-complete)
- Actual coloring is inexpensive; coalescing takes time
- Run multiple colorings & use best result (20%)
Experience suggests that there is no silver bullet
Difficulties are not symptom of a single effect

Global:
- deferred spilling
  - Briggs et al. 20%
- best of three
tenure fitch
- rematerialization
  - Briggs et al. 20%
  - expensive

→ these are good things to do
→ do not change underlying problem

- Global:
- deferred spilling
  - Briggs et al. 20%
- best of three
tenure fitch
- rematerialization
  - Briggs et al. 20%
  - expensive

→ these are good things to do
→ do not change underlying problem