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

**Strategy**

\[
\begin{align*}
il & \quad \xrightarrow{\text{select}} \quad il \\
\infty \text{ regs} & \quad \xrightarrow{\text{schedule}} \quad il \\
\infty \text{ regs} & \quad \xrightarrow{\text{allocate}} \quad il \\
\infty \text{ regs} & \quad \xrightarrow{\text{schedule}} \quad \text{asm} \\
\end{align*}
\]

*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 (and the code)
- changes storage mapping
- concern about placement of data
Definition (2 of 2)

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

Each problem is NP-complete for a non-trivial target processor.

The problems are tightly intertwined, but conventional wisdom says we can (and should) attack each one separately.
Register allocation

Concept:

\[
\begin{align*}
&\text{register allocator} \\
\downarrow \\
&\text{register code}
\end{align*}
\]

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 \( \text{loads and stores} \)
- Minimize memory space needed to hold spills
- Allocator must be efficient \( \Rightarrow \text{no backtracking} \)
Register classes

Some architectures have multiple register classes:
- *commonly*: general-purpose (GP) and floating-point (FP)
  (and double-precision may use two FPs)
- PowerPC: condition code registers
- IA 64: predicate registers, branch target registers

Problem: Interactions between classes
- If all classes were used independently, could allocate separately
- Arithmetic ops may produce condition codes, predicates, etc. to be set
- FP and other register spills may cause address arithmetic!

Our Approach
1. *Assume separate allocation for each class except where noted.*
2. *Allocate GPRs last.*
## 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><strong>Local</strong></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 NP-complete</td>
<td>adjacent register pairs $\Rightarrow$ NP-complete</td>
</tr>
<tr>
<td><strong>Global</strong></td>
<td>NP-complete for 1 register machine</td>
<td>NP-complete</td>
</tr>
<tr>
<td></td>
<td>NP-complete for $k$ register machine</td>
<td></td>
</tr>
<tr>
<td></td>
<td>most subproblems are NP-complete</td>
<td></td>
</tr>
</tbody>
</table>
Register Allocation Preliminaries

Definitions and observations

1. **Spill** virtual register (aka value) $V \equiv$
   - 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
   - must be able to compute addresses, load, & store
   - requires a minimal number of registers, $F : \text{feasible set}$
   - $F$ depends on architecture

3. $\text{MAXLIVE} \equiv$ Maximum number of values live at any instruction (in some region of code, e.g., a basic block)
Two Approaches for Single Basic Block

Common features of single-basic block allocation

- All values live in memory between basic blocks
- Therefore, even for values allocated a physical register:
  - Load before first use in block
  - Store after last def in block
- If \( \text{MAXLIVE} \leq k \), allocation is trivial; \( \mathcal{F} \) irrelevant
- If \( \text{MAXLIVE} > k \), must spill some values to memory

(1): Top-down allocation: Usage Counts

- A register can hold only a single value in entire block
- Sort variables by \( |\text{uses}| \)
- Assign first \( k - \mathcal{F} \) names to registers
- Spill all other values (load before each use; store after each def)
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

*A 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 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
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:
    \[ x_3 \leftarrow \phi(x_1, \ldots, x_n) \]
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:
   \[ X = \phi(Y, Z) : \text{Merge the sets currently holding } Y \text{ and } Z \text{ into the set holding } X \]
3. Finally, treat all references in the same live range as a single virtual register
The concept of interference

Definition of Interference

Idea: Two values cannot be in one register if “used in overlapping intervals”

Definition (interference): \( n_i \) 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
Building the interference graph

**Algorithm:**

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

\texttt{mov } v_i \rightarrow v_j

\textit{Coalesce if:}
1. \(v_i\) and \(v_j\) do not interfere, \textbf{OR}
2. \(v_i\) and \(v_j\) are \textbf{not modified} after the copy
   i.e., values remain equal always

\textit{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:

**Load / store:**
1. address computation
2. memory operation
3. estimated execution frequency

**Rematerialization:**
1. recomputing the value
2. estimated execution frequency

**Address computation:**
- minimize by keeping spilled values in activation record
- can load / store with (fp + offset) address

**Execution frequencies:**
- Static estimation:
  - weight by $10^d$ for loop depth $d$
  - weight by branching probability if enclosed in branches
- Profiling:
  - 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 $\text{degree} < k$ is $k$-colorable iff the graph with node $n$ removed is $k$-colorable

   **Proof:**
   
   ⇒ obvious
   
   ⇐ 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 $\geq k$:**
   
   🥉 Neighbors of $N$ may still use fewer than $k$ distinct colors
   
   🥉 ⇒ defer spilling until you are sure it is needed

**Idea:** Simplify graph by removing nodes

Repeat until graph is empty:

- Repeatedly remove a node with $\text{degree} < k$ from the graph
- When no such node exists: choose a candidate to spill and remove it
Algorithm

while $N$ is non-empty
    if $\exists$ node $n$ with $n^\circ < k$, 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)

Rewrite code to use assigned physical registers

*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)

Renumber

Find live ranges and rename them

Build

Build the interference graph, $I = (N, E)$

Coalesce

Fold unneeded copies:

$$l_i \rightarrow l_j \land \langle l_i, l_j \rangle \notin E \Rightarrow \text{combine } l_i \text{ & } l_j$$

Spill costs

Estimate cost for spilling each live range

Simplify

While $N$ is non-empty

If $\exists n$ with $n^o < k$, push $n$ on stack

Else, pick $n$ for possible spilling & push it

Remove $n$ from $I$

Select

While stack is non-empty

Pop $n$, insert $n$ into $I$, & try to color it

Spill

Spill uncolored definitions and uses
Picking a spill candidate

When $\forall n \in N, n^\circ \geq k$, it must pick a spill candidate

Chaitin’s heuristic

Chaitin says “minimize $\frac{\text{spill cost}}{\text{current degree}}$” where

- current degree is the number of remaining neighbors
- spill cost of $l_i$ is defined as

$$\sum_{j \in \text{refs}(l_i)} 10^{d(j)} - \sum_{\text{loads} \in \text{def}(l_i)} 2 \cdot 10^{d(j)} - \sum_{\text{stores} \in \text{use}(l_i)} 2 \cdot 10^{d(j)}$$

where

- $\text{refs}(l_i)$ is $\text{def}(l_i) \cup \text{use}(l_i)$
- $d(j)$ 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

\[
area(lr) = \sum_{i \in refs(lr)} num\_live(i) \times 5^{d(i)}
\]

<table>
<thead>
<tr>
<th>Author</th>
<th>Ratio to minimize</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chaitin</td>
<td>(\frac{cost}{current\ degree})</td>
</tr>
<tr>
<td>Bernstein</td>
<td>(\frac{cost}{current\ degree^2})</td>
</tr>
<tr>
<td>Bernstein</td>
<td>(\frac{cost}{area})</td>
</tr>
<tr>
<td>Bernstein</td>
<td>(\frac{cost}{area^2})</td>
</tr>
<tr>
<td>—</td>
<td>(\frac{cost}{area})</td>
</tr>
</tbody>
</table>

The implementation

- no metric dominates others (NP-noise)
- actual coloring is inexpensive; coalescing takes time
- run multiple colorings & use best result (20%)
How does Chaitin-style do?

Strengths and Weaknesses

↑ precise interference graph
↑ strong coalescing mechanism
↑ handles register assignment well
↑ runs relatively quickly

⇓ known to overspill in tight cases
⇓ graph has no geography
⇓ spills live range everywhere
⇓ long blocks become spilling by use counts

Is further improvement possible?

• rising spill costs
• aggressive transformations
• live range splitting
• better allocation for long blocks

Remaining problems

1. spill code
2. spill code
3. spill code

⇒ still room for improvement on this problem
Improvements

Situations that we see in practice

1. *Pass-through live ranges*
   value live but not referenced in block or region

2. *Spill the “wrong” value*
   some other value might lead to better code

3. *Excessive demand for registers*
   register pressure simply too high

*Improvements are both possible and desirable*

*Allocator is final “filter” through which code must pass*
Pass-through live ranges A regional effect

- should be lowest priority for register
- can be heavily weighted by Chaitin
- want “fair competition” inside each loop

This problem motivated Briggs’s work on live range splitting

do i ← 1 to n
   ...
   do j ← 1 to m
      do k ← 1 to o
         x ← ...
   ...
   do j ← 1 to m
      do k ← 1 to o
         no reference to x
         ...
   ...
   do j ← 1 to m
      do k ← 1 to o
         ← x
         ...
   ...

Experience suggests that there is no silver bullet

Difficulties are not symptom of a single effect

<table>
<thead>
<tr>
<th>Global:</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>deferred spilling</td>
<td>Briggs et al.</td>
<td>20%</td>
</tr>
<tr>
<td>best of three</td>
<td>Bernstein et al.</td>
<td>20%</td>
</tr>
<tr>
<td>rematerialization</td>
<td>Briggs et al.</td>
<td>20%</td>
</tr>
<tr>
<td>optimal coloring</td>
<td></td>
<td>expensive</td>
</tr>
</tbody>
</table>

→ these are good things to do
→ do not change underlying problem
## Improvements to Chaitin (2 of 2) Overview

### Regional:

<table>
<thead>
<tr>
<th>Improvement</th>
<th>Authors</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>live range splitting</td>
<td>Briggs <em>et al.</em></td>
<td>±4 ×</td>
</tr>
<tr>
<td></td>
<td>Koblenz &amp; Callahan</td>
<td>(?)</td>
</tr>
<tr>
<td>scalar replacement</td>
<td>Carr</td>
<td>20% to 3 ×</td>
</tr>
</tbody>
</table>

→ *move to near-by problem with a “better” solution*

### Local:

<table>
<thead>
<tr>
<th>Method</th>
<th>Author</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best’s method</td>
<td>Sheldon Best (1955)</td>
<td>naively “optimal”</td>
<td><em>et al.</em> in 65, 75, 88, 95</td>
</tr>
</tbody>
</table>

→ *good spill decisions*

→ *ignore surrounding context*