Topic 6b
Basic Back-End
Optimization

Register allocation
Reading List

- Slides: Topic 3a
- Dragon book: chapter 10
- S. Cooper: Chapter 13
- Other papers as assigned in class or homework
Focus of This Topic

• We focus on “scalar register allocation”
• Local register is straightforward (read Cooper’s Section 13.3)
• This global register allocation problem is essentially solved by graph coloring techniques:
  • Chaitin et. al. 1981, 82 (IBM)
  • Chow, Hennesy 1983 (Stanford)
  • Briggs, Kennedy 1992 (Rice)
• Register allocation for array variables in loops -- subject not discussed here
High-Level Compiler Infrastructure Needed – A Modern View

- Front end
  - Interprocedural Analysis and Optimization
  - Loop Nest Optimization and Parallelization
  - Global Optimization

Good IR

Code Generation
General Compiler Framework

- Good IPO
- Good LNO
- Good global optimization
- Good integration of IPO/LNO/OPT
- Smooth information passing between FE and CG
- Complete and flexible support of inner-loop scheduling (SWP), instruction scheduling and register allocation
A Map of Modern Compiler Platforms

GNU Compilers

IMPACT Compiler

Cydra VLIW Compiler

HP Research Compiler

Trimaran Compiler

SGI Pro Compiler

Open64 Compiler

Multiflow VLIW Compiler

Ucode Compiler

Chow/Hennessy

Multiflow VLIW Compiler

SGI Pro Compiler

- Designed for ILP/MP
- Production quality
- Open Source

Open64 Compiler

(PathScale, ORC, Osprey)

LLVM Compiler

SUIF Compiler

Ucode Compiler

Chow/Hennessy
Osprey Compiler Performance (4/3/07)

SPEC2000 C/ C++ Benchmark Comparison
Montecito 1.6GHz 4G Memory (higher is better)

- GCC4.3 at –O3
  - With additional options recommended by GCC developers
  - Two programs has runtime error using additional options
- Osprey3.1 with vanilla –O3
- The performance delta is ~10%, excluding two failing programs
Vision and Status of Open64 Today?

- People should view it as GCC with an alternative backend with great potential to reclaim the best compiler in the world.
- The technology incorporated all top compiler optimization research in 90's.
- It has regain momentum in the last three years due to Pathscale and HP's investment in robustness and performance.
- Targeted to x86, Itanium in the public repository, ARM, MIPS, PowerPC, and several other signal processing CPU in private branches.
Register Allocation

- Motivation
- Live ranges and interference graphs
- Problem formulation
- Solution methods
Motivation

- Registers much faster than memory
- Limited number of physical registers
- Keep values in registers as long as possible (minimize number of load/stores executed)
Goals of Optimized Register Allocation

1. Pay careful attention to allocating registers to variables that are more profitable to reside in registers.
2. Use the same register for multiple variables when legal to do so.
Brief History of Register Allocation

Chaitin:  Coloring Heuristic. Use the simple stack heuristic for register allocation. Spill/no-spill decisions are made during the stack construction phase of the algorithm.

ACM SIGPLAN Notices 1982

Briggs:  Finds out that Chaitin’s algorithm spills even when there are available registers. Solution: the optimistic approach: may-spill during stack construction, decide at spilling time.
<table>
<thead>
<tr>
<th>Chow-Hennessy</th>
<th>Priority-based coloring.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGPLAN</td>
<td>Integrate spilling decisions in the coloring decisions: spill a variable for a limited life range.</td>
</tr>
<tr>
<td>1984</td>
<td></td>
</tr>
<tr>
<td>ASPLOS</td>
<td>Favor dense over sparse use regions.</td>
</tr>
<tr>
<td>1990</td>
<td>Consider parameter passing convention.</td>
</tr>
<tr>
<td>Callahan:</td>
<td>Hierarchical Coloring Graph, register preference, profitability of spilling.</td>
</tr>
<tr>
<td>PLDI</td>
<td></td>
</tr>
<tr>
<td>1991</td>
<td></td>
</tr>
</tbody>
</table>
Assigning Registers to more Profitable Variables (example)

Source code fragment:

```c
struct c = 'S';
    sum = 0;
    i = 1;
while (i <= 100) {
    sum = sum + i;
    i = i + 1;
}
square = sum * sum;
print c, sum, square;
```
c = 'S';
sum = 0;
i = 1;
while ( i <= 100 ) {
    sum = sum + i;
i = i + 1;
}
square = sum * sum;
print c, sum, square;
For Example

Assume that there are only two non-reserved registers available for allocation ($t2$ and $t3$). A desired register allocation for the above example is as follows:

```plaintext
Variable        Register
---------------  ------------
c             no register
sum            $t2$
i             $t3$
square         $t3$
```

```plaintext
c = 'S';
sum = 0;
i = 1;
while ( i <= 100 ) {
    sum = sum + i;
i = i + 1;
}
square = sum * sum;
print c, sum, square;
```
Register Allocation Goals

1. Pay careful attention to assigning registers to variables that are more profitable

The number of defs (writes) and uses (reads) to the variables in this sample program is as follows:

```python
c = 'S';
sum = 0;
i = 1;
while ( i <= 100 ) {
    sum = sum + i;
i = i + 1;
}
square = sum * sum;
print c, sum, square;
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>#def's</th>
<th>#use's</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>sum</td>
<td>101</td>
<td>103</td>
</tr>
<tr>
<td>i</td>
<td>101</td>
<td>301</td>
</tr>
<tr>
<td>square</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

⇒ variables sum and i should get priority over variable c for register assignment.
Register Allocation Goals

2. Use the same register for multiple variables when legal to do so

⇒ Reuse same register ($t3) for variables I and square since there is no point in the program where both variables are simultaneously live.

```c

    c = 'S';
    sum = 0;
    i = 1;
    while ( i <= 100 ) {
        sum = sum + i;
        i = i + 1;
    }
    square = sum * sum;
    print c, sum, square;
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>no register</td>
</tr>
<tr>
<td>sum</td>
<td>$t2$</td>
</tr>
<tr>
<td>i</td>
<td>$t3$</td>
</tr>
<tr>
<td>square</td>
<td>$t3$</td>
</tr>
</tbody>
</table>

2008/4/18  
\course\cpeg421-08s\Topic-6b.ppt  18
Register Allocation vs. Register Assignment

Register Allocation – determining which values should be kept in registers. It ensures that the code will fit the target machine’s register set at each instruction.

Register Assignment – how to assign the allocated variables to physical registers. It produces the actual register names required by the executable code.
Local and Global Register Allocation

- Local register allocation (within a basic block): algorithms are generally straightforward – but implementation needs care [Cooper: 13.3]
- Global register allocation – graph coloring method
Liveness

Intuitively a variable \( v \) is \textit{live} if it holds a value that may be needed in the future. In other words, \( v \) is live at a point \( p_i \) if:

(i) \( v \) has been defined in a statement that precedes \( p_i \) in any path, and

(ii) \( v \) may be used by a statement \( s_j \), and there is a path from \( p_i \) to \( s_j \).

(iii) \( v \) is not killed between \( p_i \) and \( s_j \).
### Live Variables

<table>
<thead>
<tr>
<th>a: ( s1 = ld(x) )</th>
<th>s1</th>
</tr>
</thead>
<tbody>
<tr>
<td>b: ( s2 = s1 + 4 )</td>
<td>s2</td>
</tr>
<tr>
<td>c: ( s3 = s1 \times 8 )</td>
<td></td>
</tr>
<tr>
<td>d: ( s4 = s1 - 4 )</td>
<td></td>
</tr>
<tr>
<td>e: ( s5 = s1/2 )</td>
<td></td>
</tr>
<tr>
<td>f: ( s6 = s2 \times s3 )</td>
<td></td>
</tr>
<tr>
<td>g: ( s7 = s4 - s5 )</td>
<td></td>
</tr>
<tr>
<td>h: ( s8 = s6 \times s7 )</td>
<td></td>
</tr>
</tbody>
</table>

Variables **s1** and **s2** have a live range of four statements.

A variable v is live between the point \( p_i \) that succeeds its definition and the point \( p_j \) that succeeds its last use.

The interval \([p_i, p_j]\) is the **live range** of the variable v.

Which variables have the longest live range in the example?
Register Allocation

<table>
<thead>
<tr>
<th>a: s1 = ld(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>b: s2 = s1 + 4</td>
</tr>
<tr>
<td>c: s3 = s1 * 8</td>
</tr>
<tr>
<td>d: s4 = s1 - 4</td>
</tr>
<tr>
<td>e: s5 = s1/2</td>
</tr>
<tr>
<td>f: s6 = s2 * s3</td>
</tr>
<tr>
<td>g: s7 = s4 - s5</td>
</tr>
<tr>
<td>h: s8 = s6 * s7</td>
</tr>
</tbody>
</table>

How can we find out what is the minimum number of registers required by this basic block to avoid spilling values to memory?

We have to compute the live range of all variables and find the “fatest” statement (program point).

Which program points have the most variables that are live simultaneously?
Register Allocation

a: s1 = ld(x)  

b: s2 = s1 + 4

c: s3 = s1 * 8

d: s4 = s1 - 4

e: s5 = s1/2

f: s6 = s2 * s3

g: s7 = s4 - s5

h: s8 = s6 * s7

At statement e variables s1, s2, s3, and s4 are live, and during statement f variables s2, s3, s4, and s5 are live.

But we have to use some math: our choice is liveness analysis.
Live-in and Live-out

\[ a: \text{s1} = \text{ld(x)} \]
\[ b: \text{s2} = \text{s1} + 4 \]
\[ c: \text{s3} = \text{s1} \times 8 \]
\[ d: \text{s4} = \text{s1} - 4 \]
\[ e: \text{s5} = \text{s1}/2 \]
\[ f: \text{s6} = \text{s2} \times \text{s3} \]
\[ g: \text{s7} = \text{s4} - \text{s5} \]
\[ h: \text{s8} = \text{s6} \times \text{s7} \]

**live-in(r):** set of variables that are live at the point that immediately precedes statement \( r \).

**live-out(r):** set of variables that are live at the point that immediately succeeds \( r \).
Live-in and Live-out: Program Example

a: \( s_1 = \text{ld}(x) \)

b: \( s_2 = s_1 + 4 \)

c: \( s_3 = s_1 \times 8 \)

d: \( s_4 = s_1 - 4 \)

e: \( s_5 = s_1/2 \)

f: \( s_6 = s_2 \times s_3 \)

g: \( s_7 = s_4 - s_5 \)

h: \( s_8 = s_6 \times s_7 \)

What are live-in(e) and live-out(e)?

live-in(e) = \{s_1, s_2, s_3, s_4\}  live-out(e) = \{s_2, s_3, s_4, s_5\}
Live-in and Live-out in Control Flow Graphs

live-in(B): set of variables that are live at the point that immediately precedes the first statement of the basic block B.

live-out(B): set of variables that are live at the point that immediately succeeds the last statement of the basic block B.
Live-in and Live-out of basic blocks

- \text{live-in}(B_1) = \{b, c, d, f\}
- \text{live-in}(B_2) = \{a, c, d, e\}
- \text{live-in}(B_3) = \{a, c, d, f\}
- \text{live-in}(B_4) = \{c, d, f\}

- \text{live-out}(B_1) = \{a, c, d, e, f\}
- \text{live-out}(B_2) = \{c, d, e, f\}
- \text{live-out}(B_3) = \{b, c, d, e, f\}
- \text{live-out}(B_4) = \{b, c, d, e, f\}

\text{b, d, e, f live}

\text{b, c, d, e, f live}
A register-interference graph is an undirected graph that summarizes live analysis at the variable level as follows:

- A node is a variable/temporary that is a candidate for register allocation (exceptions are volatile variables and aliased variables).
- An edge connects nodes V1 and V2 if there is some program point in the program where variables V1 and V2 are live simultaneously. (Variables V1 and V2 are said to interfere, in this case.)
Register Interference Graph: Program Example

- a: $s1 = ld(x)$
- b: $s2 = s1 + 4$
- c: $s3 = s1 \times 8$
- d: $s4 = s1 - 4$
- e: $s5 = s1/4$
- f: $s6 = s2 \times s3$
- g: $s7 = s4 - s5$
- h: $s8 = s6 \times s7$
Local Register Allocation vs. Global Register Allocation

- **Local Register Allocation (basic block level)**
  - Allocate for a single basic block - using liveness information
  - generally straightforward
  - may not need graph coloring

- **Global Register Allocation (CFG)**
  - Allocate among basic blocks
  - graph coloring method
  - Need to use global liveness information
Register Allocation by Graph Coloring

Background: A graph is said to be k-colored if each node has been assigned one of k colors in such a way that no two adjacent nodes have the same color.

Basic idea: A k-coloring of the interference graph can be directly mapped to a legal register allocation by mapping each color to a distinct register. The coloring property ensures that no two variables that interfere with each other are assigned the same register.
The basic idea behind register allocation by graph coloring is to

1. Build the register interference graph,

2. Attempt to find a k-coloring for the interference graph.
Complexity of the Graph Coloring Problem

- The problem of determining if an undirected graph is k-colorable is NP-hard for $k \geq 3$.
- It is also hard to find approximate solutions to the graph coloring problem.
**Register Allocation**

**Question:** What to do if a register-interference graph is not k-colorable? Or if the compiler cannot efficiently find a k-coloring even if the graph is k-colorable?

**Answer:** Repeatedly select less profitable variables for “spilling” (i.e. not to be assigned to registers) and remove them from the interference graph till the graph becomes k-colorable.
Estimating Register Profitability

The register profitability of variable $v$ is estimated by:

$$profitability(v) = \sum_{i} freq(i) \cdot savings(v, i)$$

$freq(i)$: estimated execution frequency of basic block $i$ (obtained by profiling or by static analysis),

$savings(v, i)$: estimated number of processor cycles that would be saved due to a reduced number of load and store instructions in basic block $i$, if a register was assigned to variable $v$. 
Example of Estimating Register Profitability

Basic block frequencies for previous example:

<table>
<thead>
<tr>
<th>B</th>
<th>freq(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[100]</td>
<td>1</td>
</tr>
<tr>
<td>[101]</td>
<td>1</td>
</tr>
<tr>
<td>[102]</td>
<td>1</td>
</tr>
<tr>
<td>[103]</td>
<td>101</td>
</tr>
<tr>
<td>[104]</td>
<td>101</td>
</tr>
<tr>
<td>[105]</td>
<td>100</td>
</tr>
<tr>
<td>[106]</td>
<td>100</td>
</tr>
<tr>
<td>[107]</td>
<td>100</td>
</tr>
<tr>
<td>[108]</td>
<td>1</td>
</tr>
<tr>
<td>[109]</td>
<td>1</td>
</tr>
<tr>
<td>[110]</td>
<td>1</td>
</tr>
</tbody>
</table>
Estimation of Profitability

(Assume that load and store instructions take 1 cycle each on the target processor)

\[
\text{Profitability(c)} = \text{freq}([100]) \times (1 - 0) + \text{freq}([110]) \times (1 - 0) \\
= 2
\]

\[
\text{Profitability(sum)} = \text{freq}([101]) \times (1 - 0) + \text{freq}([105]) \times (2 - 0) \\
+ \text{freq}([109]) \times (2 - 0) \\
= 1 \times 1 + 100 \times 2 + 1 \times 2 = 203
\]

\[
\text{Profitability(i)} = \text{freq}([102]) \times (1 - 0) + \text{freq}([104]) \times (1 - 0) \\
+ \text{freq}([105]) \times (1 - 0) + \text{freq}([106]) \times (2 - 0) \\
= 1 \times 1 + 101 \times 1 + 100 \times 1 + 100 \times 2 = 402
\]

\[
\text{Profitability(square)} = \text{freq}([109]) \times (1 - 0) + \text{freq}([110]) \times (1 - 0) \\
= 2
\]
Heuristic Solutions

Key observation:

<table>
<thead>
<tr>
<th>G</th>
<th>Remove a node x with degree &lt; k</th>
</tr>
</thead>
<tbody>
<tr>
<td>G'</td>
<td>From G, and all associated edge</td>
</tr>
</tbody>
</table>

What do we know for G’s k-colorability if we know G’ is k-colorable?

- **Answer:** If G’ is k-colorable => So is G!
A 2-Phase Register Allocation Algorithm

Build IG -> Simplify -> Select and Spill

Forward pass

Reverse pass
Heuristic “Optimistic” Algorithm

/* Build step */
Build the register-interference graph, G;

/* Forward pass */
Initialize an empty stack;
repeat
  while G has a node v such that |neighbor(v)| < k do
    /* Simplify step */
    Push (v, no-spill)
    Delete v and its edges from G
  end while

if G is non-empty then
  /* Spill step */
  Choose “least profitable” node v as a potential spill node;
  Push (v, may-spill)
  Delete v and its edges from G
end if
until G is an empty graph;
Heuristic “Optimistic” Algorithm

/* Reverse Pass */
while the stack is non-empty do
    Pop (v, tag)
    N := set of nodes in neighbors(v);
    if (tag = no-spill) then
        /* Select step */
        Select a register R for v such that
            R is not assigned to nodes in N;
        Insert v as a new node in G;
        Insert an edge in G
            from v to each node in N;
    else /* tag = may-spill */
        if v can be assigned a register R
            such that R is not assigned
                to nodes in N then
            /* Optimism paid off: need not spill */
            Assign register R to v;
            Insert v as a new node in G;
            Insert an edge in G
                from v to each node in N;
        else /* Need to spill v */
            Mark v as not being allocate a register
        end if
    end if
end while
Remarks

The above register allocation algorithm based on graph coloring is both efficient (linear time) and effective.

It has been used in many industry-strength compilers to obtain significant improvements over simpler register allocation heuristics.
Extensions

• Coalescing

• Live range splitting
Coalescing

In the sequence of intermediate level instructions with a copy statement below, assume that registers are allocated to both variables x and y.

```
  x := ...
  ...
  y := x
  ...
  ... := y
```

There is an opportunity for further optimization by eliminating the copy statement if x and y are assigned the same register.

The constraint that x and y receive the same register can be modeled by coalescing the nodes for x and y in the interference graph i.e., by treating them as the same variable.
An Extension with Coalesce

- Build IG
- Simplify
- Coalesce
- Select and Spill
Register Allocation with Coalescing

1. **Build**: build the register interference graph $G$ and categorize nodes as *move-related* or *non-move-related*.

2. **Simplify**: one at a time, remove non-move-related nodes of low ($< K$) degree from $G$.

3. **Coalesce**: conservatively coalesce $G$: only coalesce nodes $a$ and $b$ if the resulting $a$-$b$ node has less than $K$ neighbors.

4. **Freeze**: If neither coalesce nor simplify works, freeze a move-related node of low degree, making it non-move-related and available for simplify.

(Appel, pp. 240)
5. **Spill**: if there are no low-degree nodes, select a node for potential spilling.

6. **Select**: pop each element of the stack assigning colors.
Example: Step 1: Compute Live Ranges

LIVE-IN: k j

\[ g \leftarrow \text{mem}[j+12] \]
\[ h \leftarrow k - 1 \]
\[ f \leftarrow g + h \]
\[ e \leftarrow \text{mem}[j+8] \]
\[ m \leftarrow \text{mem}[j+16] \]
\[ b \leftarrow \text{mem}[f] \]
\[ c \leftarrow e + 8 \]
\[ d \leftarrow c \]
\[ k \leftarrow m + 4 \]
\[ j \leftarrow b \]

LIVE-OUT: d k j

\[ k \quad j \]
\[ g \quad h \]
\[ f \]
\[ e \quad m \]
\[ b \quad c \]
\[ d \]
Example:
Step 3: Simplify (K=4)

(Appel, pp. 237)
Example: Step 3: Simplify (K=4)

stack
(g, no-spill)
(h, no-spill)

(Appel, pp. 237)
Example:
Step 3: Simplify (K=4)

(k, no-spill)
(g, no-spill)
(h, no-spill)

(Appel, pp. 237)
Example:
Step 3: Simplify (K=4)

Stack:
(f, no-spill)
(k, no-spill)
(g, no-spill)
(h, no-spill)

(Appel, pp. 237)
Example:
Step 3: Simplify (K=4)

stack
(e, no-spill)
(f, no-spill)
(k, no-spill)
(g, no-spill)
(h, no-spill)

(Appel, pp. 237)
Example: Step 3: Simplify (K=4)

(j, no-spill) (d, no-spill) (b, no-spill) (m, no-spill)
Example:
Step 3: Coalesce \( (K=4) \)

Why we cannot simplify?

Cannot simplify move-related nodes.

(Appel, pp. 237)
Example: Step 3: Coalesce (K=4)

```
Example: Step 3: Coalesce (K=4)

stack

(m, no-spill)
(e, no-spill)
(f, no-spill)
(k, no-spill)
(g, no-spill)
(h, no-spill)
```

Appel, pp. 237
Example:
Step 3: Simplify \((K=4)\)

\[\text{stack} \]
(c-d, no-spill)
(m, no-spill)
(e, no-spill)
(f, no-spill)
(k, no-spill)
(g, no-spill)
(h, no-spill)

(Appel, pp. 237)
Example: Step 3: Coalesce (K=4)

stack
(c-d, no-spill)
(m, no-spill)
(e, no-spill)
(f, no-spill)
(k, no-spill)
(g, no-spill)
(h, no-spill)

(Appel, pp. 237)
Example: Step 3: Simplify \((K=4)\)

- Stack:
  - \((b-j, \text{no-spill})\)
  - \((c-d, \text{no-spill})\)
  - \((m, \text{no-spill})\)
  - \((e, \text{no-spill})\)
  - \((f, \text{no-spill})\)
  - \((k, \text{no-spill})\)
  - \((g, \text{no-spill})\)
  - \((h, \text{no-spill})\)

- Stack:
  - \((b-j, \text{no-spill})\)

(Appel, pp. 237)
Example:
Step 3: Select (K=4)

(b-j, no-spill)
(c-d, no-spill)
(m, no-spill)
(e, no-spill)
(f, no-spill)
(k, no-spill)
(g, no-spill)
(h, no-spill)

(Appel, pp. 237)
Example:

Step 3: Select (K=4)

stack
(b-j, no-spill)
(c-d, no-spill)
(m, no-spill)
(e, no-spill)
(f, no-spill)
(k, no-spill)
(g, no-spill)
(h, no-spill)
Example: Step 3: Select (K=4)

(b-j, no-spill)
(c-d, no-spill)
(m, no-spill)
(e, no-spill)
(f, no-spill)
(k, no-spill)
(g, no-spill)
(h, no-spill)

(Appel, pp. 237)
Example: Step 3: Select (K=4)

Stack

(b-j, no-spill)
(c-d, no-spill)
(m, no-spill)
(e, no-spill)
(f, no-spill)
(k, no-spill)
(g, no-spill)
(h, no-spill)

(Appel, pp. 237)
Example: Step 3: Select (K=4)

(b-j, no-spill)
(c-d, no-spill)
(m, no-spill)
(e, no-spill)
(f, no-spill)
(k, no-spill)
(g, no-spill)
(h, no-spill)

(Appel, pp. 237)
Example:
Step 3: Select \((K=4)\)

![Diagram of a directed graph with nodes labeled j, k, b, m, d, e, f, and g, and edges connecting them. The stack contains the following no-spill instructions: (b-j, no-spill), (c-d, no-spill), (m, no-spill), (e, no-spill), (f, no-spill), (k, no-spill), (g, no-spill), (h, no-spill).]

(Appel, pp. 237)
Example:
Step 3: Select (K=4)

Stack

(b-j, no-spill)
(c-d, no-spill)
(m, no-spill)
(e, no-spill)
(f, no-spill)
(k, no-spill)
(g, no-spill)
(h, no-spill)

(Appel, pp. 237)
Example:
Step 3: Select (K=4)

The diagram shows a graph with nodes labeled j, k, b, m, f, e, d, c, and g. The nodes are connected by lines, indicating dependencies or relationships. The stack is highlighted with the following entries:

- (b-j, no-spill)
- (c-d, no-spill)
- (m, no-spill)
- (e, no-spill)
- (f, no-spill)
- (k, no-spill)
- (g, no-spill)
- (h, no-spill)

(Appel, pp. 237)
Live Range Splitting

The basic coloring algorithm does not consider cases in which a variable can be allocated to a register for part of its live range.

Some compilers deal with this by splitting live ranges within the iteration structure of the coloring algorithm i.e., by pretending to split a variable into two new variables, one of which might be profitably assigned to a register and one of which might not.
Length of Live Ranges

The interference graph *does not* contain information of where in the CFG variables interfere and what the length of a variable’s live range is. For example, if we only had few available registers in the following intermediate-code example, the right choice would be to spill variable w because it has the longest live range:

\[
\begin{align*}
  x &= w + 1 \\
  c &= a - 2 \\
  \ldots \ldots \\
  y &= x \times 3 \\
  z &= w + y
\end{align*}
\]
Effect of Instruction Reordering on Register Pressure

The coloring algorithm does not take into account the fact that reordering IL instructions can reduce interference. Consider the following example:

Original Ordering
(needs 3 registers)

- \( t_1 := A[i] \)
- \( t_2 := A[j] \)
- \( t_3 := A[k] \)
- \( t_4 := t_2 * t_3 \)
- \( t_5 := t_1 + t_4 \)

Optimized Ordering
(needs 2 registers)

- \( t_2 := A[j] \)
- \( t_3 := A[k] \)
- \( t_4 := t_2 * t_3 \)
- \( t_1 := A[i] \)
- \( t_5 := t_1 + t_4 \)