Feb 12 — Global Register Allocation
Review: Dominance Frontier

Dominance Frontiers
• $DF(n)$ is fringe just beyond the region $n$ dominates
• $m \in DF(n)$ : iff $n \notin (\text{Dom}(m) - \{m\})$ but $n \in \text{Dom}(p)$ for some $p \in \text{preds}(m)$.

i.e., $n$ dominates $p$

i.e., $n$ doesn’t strictly dominate $m$

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dom</td>
<td>0</td>
<td>0,1</td>
<td>0,1,2</td>
<td>0,1,3</td>
<td>0,1,3,4</td>
<td>0,1,3,5</td>
<td>0,1,3,6</td>
<td>0,1,7</td>
</tr>
<tr>
<td>DF</td>
<td>–</td>
<td>–</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>1</td>
</tr>
</tbody>
</table>
Review: Practice Example 1

- IDOM  DF phi renaming reach deconstruct

A:

\[
\begin{align*}
a &= 1
\end{align*}
\]

B:

\[
\begin{align*}
a &= 2
\end{align*}
\]

C:

\[
\begin{align*}
b &= a
\end{align*}
\]
Review: Practice Example 2

- IDOM
- DF
- phi
- renaming
- reach
- deconstruct

A: \[ a = 1 \]

B: \[ a = a + 1 \]
Compiler Back-end

Responsibilities

- Translate IR into target machine code
- Choose instruction to implement each IR operation
- Decide which value to keep in registers
- Ensure conformance with system interface

Many of the backend problems are NP-complete
Register Allocation

- Produce correct code that uses $k$ (or fewer) registers
- Minimize the cost of spilling: cycles due to added loads & stores
- Minimize space used to hold spilled values
An Example

Here is a sample code sequence in SSA form

1. loadI 1028 => r1
2. load r1 => r2
3. mult r1, r2 => r3
4. loadI 5 => r4
5. sub r4, r2 => r5
6. loadI 8 => r6
7. mult r5, r6 => r7
8. sub r7, r3 => r8
9. store r8 => r1
An Example

- Here is a sample code sequence in SSA form

```plaintext
1. loadI 1028 => r1 // r1
2. load r1 => r2  // r1 r2
3. mult r1, r2 => r3 // r1 r2 r3
4. loadI 5 => r4  // r1 r2 r3 r4
5. sub r4, r2 => r5 // r1 r3 r5
6. loadI 8 => r6  // r1 r3 r5 r6
7. mult r5, r6 => r7 // r1 r3 r7
8. sub r7, r3 => r8 // r1 r8
9. store r8 => r1 //
```

**Live-on-exit set:** the set of variables that are live on the exit of an instruction

**Live-on-entry set:** the set of variables that are live on the exit of an instruction
An Example

• Here is a sample code sequence in SSA form

1  loadI 1028 => r1 // r1
2  load r1 => r2 // r1 r2
3  mult r1, r2 => r3 // r1 r2 r3
4  loadI 5 => r4 // r1 r2 r3 r4
5  sub r4, r2 => r5 // r1 r3 r5
6  loadI 8 => r6 // r1 r3 r5 r6
7  mult r5, r6 => r7 // r1 r3 r7
8  sub r7, r3 => r8 // r1 r8
9  store r8 => r1 //

• Interference graph $G_I = (N, E_I)$

* Nodes in $G_I$ represent values, or live ranges
* Edges in $G_I$ represent individual interferences
  
  For $x, y \in N$, $(x, y) \in E_I$ iff $x$ and $y$ interfere

* A $k$-coloring of $G_I$ can be mapped to an allocation mapped to $k$ registers

“interference”: two variables interfere if there exists an operation where both are simultaneously live

Two interfering variables cannot occupy the same register
Graph Coloring

• Here is a sample code sequence in SSA form

```plaintext
1 loadI 1028 => r1 // r1
2 load r1 => r2 // r1 r2
3 mult r1, r2 => r3 // r1 r2 r3
4 loadI 5 => r4 // r1 r2 r3 r4
5 sub r4, r2 => r5 // r1 r3 r5
6 loadI 8 => r6 // r1 r3 r5 r6
7 mult r5, r6 => r7 // r1 r3 r7
8 sub r7, r3 => r8 // r1 r8
9 store r8 => r1 //
```

• Color the interference graph

* A graph G is said to be k-colorable iff the nodes can be labeled with integers 1 ... k so that no edge in G connects two nodes with the same label
* Each color can be mapped to a distinct physical register

“interference”: two variables interfere if there exists an operation where both are simultaneously live

Two interfering variables (node) cannot occupy the same register (color)
Graph Coloring

• Here is a sample code sequence in SSA form

```plaintext
1  loadI 1028 => r1   // r1
2  load  r1 => r2     // r1 r2
3  mult  r1, r2   => r3 // r1 r2 r3
4  loadI  5   => r4   // r1 r2 r3 r4
5  sub   r4, r2   => r5 // r1      r3      r5
6  loadI  8   => r6   // r1      r3      r5 r6
7  mult  r5, r6   => r7 // r1      r3                r7
8  sub   r7, r3   => r8 // r1                      r8
9  store r8     => r1 //
```

• Color the interference graph

* A graph G is said to be $k$-colorable iff the nodes can be labeled with integers 1 … k so that no edge in G connects two nodes with the same label
* Each color can be mapped to a distinct physical register

In this case, the graph is 4-colorable
Building Interference Graph

- Two values *interfere* if there exits an operation where both are simultaneously live.
- To compute interference graph, we must know where values are “live.”

```
1. loadI 1028 => r1       // r1
2. load r1      => r2     // r1 r2
3. mult r1, r2   => r3    // r1 r2 r3
4. loadI 5       => r4    // r1 r2 r3 r4
5. sub r4, r2    => r5    // r1 r3 r5
6. loadI 8       => r6    // r1 r3 r5 r6
7. mult r5, r6   => r7    // r1 r3 r7
8. sub r7, r3    => r8    // r1 r8
9. store r8      => r1
```

Let’s start from local interference graph within a basic block:

*backward traversal — instruction by instruction*
Building Global Interference Graph

- Discover live ranges
  * Construct the **SSA form** of the procedure
  * At each $\Phi$-function, take the union of the arguments
  * Rename to reflect these new “live ranges”

- Compute LIVE sets over live ranges for each basic block
  * Solve equations for LIVE over domain of live range names
  * Use a simple iterative data-flow solver

- Iterate over each block, from bottom to top
  * Construct **LIVENOW** at each point in the block, in a backward traversal
  * At each operation, add appropriate edges to the graph & update LIVENOW
    -> Add an edge from result to each value in LIVE
    -> Remove result from LIVE
    -> Add each operand to LIVE

} update the LIVENOW set
Building Global Interference Graph

- Discover live ranges
  * Construct the SSA form of the procedure
  * At each $\Phi$-function, take the union of the arguments
  * Rename to reflect these new “live ranges”

- Compute LIVE sets over live ranges for each basic block
  * Solve equations for LIVE over domain of live range names
  * Use a simple iterative data-flow solver

- Iterate over each block, from bottom to top
  * Construct LIVENOW at each point in the block, in a backward traversal
  * At each operation, add appropriate edges to the graph & update LIVENOW
    -> Add an edge from result to each value in LIVE
    -> Remove result from LIVE
    -> Add each operand to LIVE

} update the LIVENOW set
**LIVE RANGES**

In the multi-block case, live ranges are more complex than in the local case.

- Consider $x$, $y$, & $z$ in the code to the right.

```
B_0  x ← ...
     z ← ...

B_1  y ← ...

B_2  x ← y
     y ← x
     x ← ...

B_3  ... ← y
     y ← z

B_4  ... ← ...
     z ← ...

B_5  ... ← x + y + z
```
**LIVE RANGES**

In the multi-block case, live ranges are more complex than in the local case.

- Consider $x$, $y$, & $z$ in the code to the right
  - $x$ has 2 distinct live ranges
LIVE RANGES

In the multi-block case, live ranges are more complex than in the local case.

- Consider x, y, & z in the code to the right
  - x has 2 distinct live ranges
  - y has 2 distinct live ranges
LIVE RANGES

In the multi-block case, live ranges are more complex than in the local case.

- Consider x, y, & z in the code to the right
  - x has 2 distinct live ranges
  - y has 2 distinct live ranges
  - z has just 1 live range
    → z is never live in $B_2$
- Finding live ranges takes some work
FINDING LIVE RANGES

We can use SSA form to find live ranges in a simple way
1. Build static single assignment form (SSA form)
2. Consider each SSA name a set
3. At each phi-function, union together the sets of the phi-function arguments
4. Each remaining set is a live range
5. Rename into live ranges
LIVE RANGES

Example in (Pruned) SSA Form

• Each name is defined in exactly one operation
• Each use refers to one name
• Live ranges are
  ✶ $(x_0, x_2, x_3)$ and $(x_1)$
  ✶ $(y_0)$ and $(y_1, y_2, y_3)$
  ✶ $(z_0, z_1, z_2)$

as predicted several slides ago
LIVE RANGES

Rename Around Live Ranges
- Go back to original (non-SSA code) & name each live range
- Live ranges are
  - $(x_0, x_2, x_3)$ and $(x_1)$
  - $(y_0)$ and $(y_1, y_2, y_3)$
  - $(z_0, z_1, z_2)$
  as predicted several slides ago
- Note that a copy operation, such as $x \leftarrow y$ does not place $x$ and $y$ in the same live range

Copyright 2013. Keith D. Cooper & Linda Torczon. Rice University
Building Global Interference Graph

- Discover live ranges
  * Construct the **SSA form** of the procedure
  * At each Φ-function, take the union of the arguments
  * Rename to reflect these new “live ranges”

- Compute LIVE sets over live ranges for each basic block
  * Solve equations for LIVE over domain of live range names
  * Use a simple iterative data-flow solver

- Iterate over each block, from bottom to top
  * Construct **LIVENOW** at each point in the block, in a backward traversal
  * At each operation, add appropriate edges to the graph & update LIVENOW
    -> Add an edge from result to each value in LIVE
    -> Remove result from LIVE
    -> Add each operand to LIVE
    } update the LIVENOW set
Building Global Interference Graph

- Discover live ranges
  * Construct the SSA form of the procedure
  * At each Φ-function, take the union of the arguments
  * Rename to reflect these new “live ranges”

- Compute LIVE sets over live ranges for each basic block
  * Solve equations for LIVE over domain of live range names
  * Use a simple iterative data-flow solver

- Iterate over each block, from bottom to top
  * Construct LIVENOW at each point in the block, in a backward traversal
  * At each operation, add appropriate edges to the graph & update LIVENOW
    -> Add an edge from result to each value in LIVE
    -> Remove result from LIVE
    -> Add each operand to LIVE

} update the LIVENOW set
Computing LIVE sets

- Iterative data flow analysis
  * LIVE set definition (assuming we use SSA form)
    -> LIVE\_beg(n): the set of values that are live at the beginning of basic block n
    -> USE(n): the set of values that are used in basic block n
    -> DEF(n): the set of values that are defined in basic block n
    -> LIVE\_end(n): the set of values that are live at the end of basic block n
  * Data flow equations
    -> LIVE\_end(n) = ∪ s∈succ(n) LIVE\_beg(s)
    -> LIVE\_beg(n) = { USE(n) | UNION LIVE\_end(n) } ∩ DEF(n)
  * Initialization
    -> LIVE\_end(s\_leave) = {}  
  * Convergence
    -> Fixed point algorithm
    -> Convergence after a limited number of steps
Computing LIVE sets

- Iterative data flow analysis
  * LIVE set definition (assuming we use SSA form)
    - \( \text{LIVE}_\text{beg}(n) \): the set of values that are live at the beginning of basic block \( n \)
    - \( \text{USE}(n) \): the set of values that are used in basic block \( n \)
    - \( \text{DEF}(n) \): the set of values that are defined in basic block \( n \)
    - \( \text{LIVE}_\text{end}(n) \): the set of values that are live at the end of basic block \( n \)
  * Data flow equations
    - \( \text{LIVE}_\text{end}(n) = \bigcup_{s \in \text{succ}(n)} \text{LIVE}_\text{beg}(s) \)
    - \( \text{LIVE}_\text{beg}(n) = \{ \text{USE}(n) \} \cup \text{LIVE}_\text{end}(n) \cap \text{DEF}(n) \)
  * Initialization
    - \( \text{LIVE}_\text{end}(s_{\text{leave}}) = \{ \} \)
  * Convergence
    - Fixed point algorithm
    - Convergence after a limited number of steps
Building Global Interference Graph

- Discover live ranges
  * Construct the **SSA form** of the procedure
  * At each Φ-function, take the union of the arguments
  * Rename to reflect these new “live ranges”

- Compute LIVE sets over live ranges for each basic block
  * Solve equations for LIVE over domain of live range names
  * Use a simple iterative data-flow solver

- Iterate over each block, from bottom to top
  * Construct **LIVENOW** at each point in the block, in a backward traversal
  * At each operation, add appropriate edges to the graph & update LIVENOW
    -> Add an edge from result to each value in LIVE
    -> Remove result from LIVE
    -> Add each operand to LIVE
      \{ \text{LIVE}(n) \text{ is initialized as LIVE}_{\text{end}}(n) \}
"/>update the LIVENOW set
Chaitin’s Algorithm

- Suppose we have k registers — a k-coloring problem

- Observation
  * Any vertex n that fewer than k neighbors in the interference graph can always be colored!
    Pick any color not used by its neighbors — there must be one

- Chaitin’s algorithm
  * Step 1: Pick any vertex n such that deg(n) < k and put it on the stack
  * Step 2: Remove that vertex and all its incident edges from the interference graph
    -> This may make remaining nodes have fewer neighbors
  * Step 3: If there does not exist such vertex, pick one vertex m and spill.
    Remove the vertex m and all its incident edges from the graph.
    Go back to step 1.
  Repeat Step 1, 2 and 3 until no nodes are left in the graph.
  * Step 4: 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

Stack

1 is the only node with degree < 3

Copyright 2013. Keith D. Cooper & Linda Torczon. Rice University
CHAITIN’S ALGORITHM IN PRACTICE

3 Registers

Stack

Now, 2 & 3 have degree < 3
CHAITIN’S ALGORITHM IN PRACTICE

3 Registers

Stack

Now all nodes have degree < 3
CHAITIN’S ALGORITHM IN PRACTICE

3 Registers

Stack

Colors:
1:  
2:  
3:  

Copyright 2013. Keith D. Cooper & Linda Torczon. Rice University
CHAITIN’S ALGORITHM IN PRACTICE

3 Registers

Stack

Colors:
1: Yellow
2: Red
3: Blue
CHAITIN’S ALGORITHM IN PRACTICE

3 Registers

Stack

Colors:
1:  
2:  
3:  

Copyright 2013. Keith D. Cooper & Linda Torczon. Rice University
CHAITIN’S ALGORITHM IN PRACTICE

3 Registers

Stack

Colors:
1: ●
2: ●
3: ●
Chaitin-Briggs Algorithm

• Improvement over Chaitin’s algorithm

• Observation
  * A node that has more than k-1 neighbors is not necessarily un-colorable
    -> It depends the number of colors of its neighbors

• Brigg’s idea
  * Take that node as spilling candidate and still push into stack
  * When you pop it off, a color might be available for it

Maximum degree is a loose upper bound on colorability
CHAITIN-BRIGGS IN PRACTICE

2 Registers

Stack

No node has degree < 2
- Chaitin would spill a node
- Briggs picks the same node & stacks it

Copyright 2013. Keith D. Cooper & Linda Torczon. Rice University
CHAITIN-BRIGGS IN PRACTICE

2 Registers

Stack

Pick a node, say 1
CHAITIN-BRIGGS IN PRACTICE

2 Registers

Stack

Pick a node, say 1
CHAITIN-BRIGGS IN PRACTICE

2 Registers

Stack

Now, both 2 & 3 have degree < 2
Pick one, say 3
Both 2 & 4 have degree < 2.
Take them in order 2, then 4.
CHAITIN-BRIGGS IN PRACTICE

2 Registers

Stack

4
CHAITIN-BRIGGS IN PRACTICE

2 Registers

Stack

Now, rebuild the graph
CHAITIN-BRIGGS IN PRACTICE

2 Registers

Stack

Colors:
1: 
2: 

Copyright 2013. Keith D. Cooper & Linda Torczon. Rice University
CHAITIN-BRIGGS IN PRACTICE

2 Registers

Stack

Colors:
1: 
2: 

Copyright 2013. Keith D. Cooper & Linda Torczon. Rice University
CHAITIN-BRIGGS IN PRACTICE

2 Registers

Stack

Colors:
1: 🔵
2: 🔴
Comparison

**Chaitin’s Allocator**

1. renumber
2. build
3. coalesce
4. spill costs
5. simplify
6. select
7. spill

- Get live range & rename
- Build interference graph
- Copy coalescing (Kun’s presentation)
- Choose a live range to spill
- Remove nodes from graph

**Chaitin-Briggs Allocator**

1. renumber
2. build
3. coalesce
4. spill costs
5. simplify
6. select
7. spill ?

- Spill decision made at the removal phase
- Spill decision made at the coloring phase

**The difference**