CS516 Programming Languages and Compilers II

Zheng Zhang
Spring 2015

Feb 19 — Global Register Allocation II

Rutgers University
Review: 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

-> Add an edge from result to each value in LIVE
-> Remove result from LIVE
-> Add each operand to LIVE

update the LIVENOW set
Review: 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
Review: Live Ranges (Webs)

- Connect relevant definition and uses
  * Def and all reachable uses must be in the same web
  * All defs that reach the same use must be in the same web

- Example
  * Multi-block case
  * z has one live range
  * x has ? live range
  * y has ? live range
We can use SSA form
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 operands
4. Each remaining set is a live range
5. Rename into live ranges
Review: 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
Review: Computing LIVE sets

- Iterative data flow analysis
  * LIVE set definition (assuming we use SSA form)
    - $\text{USE}(n)$: the set of values whose live range span the beginning of 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
    - $\text{LIVE}_{\text{end}}(n) = \bigcup_{s \in \text{succ}(n)} \{ \text{LIVE}_{\text{end}}(s) \cap \overline{\text{DEF}}(s) \} \cup \text{USE}(s)$
  * Initialization
    - $\text{LIVE}_{\text{end}}(s_{\text{leave}}) = \{\}$
  * Convergence
    - Fixed point algorithm
    - Convergence after a limited number of steps
Review: Computing LIVE sets

- Iterative data flow analysis
  * LIVE set definition (assuming we use SSA form)
    -> USE(n): the set of values whose live range span the beginning of 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(s) = \{ LIVE_end(s) ∩ DEF(s) \} ∪ USE(s)
  * Initialization
    -> LIVE_end(s_{leaf}) = \{\}
  * Convergence
    -> Fixed point algorithm
    -> Convergence after a limited number of steps
Review: 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

LIVE(n) is initialized as LIVE_end(n)
Review: Building LIVENOW in a Basic Block

- Backwards instruction-by-instruction traversal
  1. Initialize LIVENOW for the last instruction at block $n$ as LIVE_end($n$)
  2. For every instruction $i$ in the backwards traversal, update LIVENOW
     * Update interference graph by connection result operand to every other node in LIVENOW
     * Remove result operand from LIVENOW set
     * Add source operand into LIVENOW set

```
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
```

This is also local interference graph construction algorithm!
Review: Chaitin’s Coloring 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
  1 Simplify: Pick a vertex n such that deg(n) < k and push it onto the stack
     Remove that vertex and all its incident edges from the interference graph
     -> This may make remaining nodes have fewer neighbors
  2 Spill: If no vertex has less than k neighbours, we pick one vertex m to spill.
     Remove the vertex m and all its incident edges from the graph.
     Repeat step 2 until we can go back to step 1
  3 Repeat step 1 and 2 until no vertices are left
  4 Assignment: Pop vertices out of the stack one by one. Give every vertices a color as the lowest color label that is not used by any neighbour.
Review: Chaitin’s Allocator

- Get live range & rename
- Build interference graph
- Copy coalescing: two variables associated by one MOV instruction can be coalesced
- Calculate spilling cost for every live range
- Choose a live range to spill if necessary
- If possible: remove a node from the graph - if the number of neighbors < k
- Select the node as a colorable node and push it to stack

Otherwise: choose a node to spill, remove the node and its incident edges
Review: Chaitin—Briggs Allocator

1. Get live range & rename
2. Build interference graph
3. Copy coalescing: two variables associated by one MOV instruction can be coalesced
4. Calculate spilling cost for every live range
5. Choose a live range to spill if necessary
6. If possible: remove a node from the graph - if the number of neighbors < k
7. Select the node as a colorable node and push it to stack

Otherwise: choose a node to spill, remove the node and its incident edges. **But still push the node into stack.**
Review: Chaitin—Briggs Allocator

why rebuild interference graph?

- Get live range & rename
- Build interference graph
- Copy coalescing: two variables associated by one MOV instruction can be coalesced
- Calculate spilling cost for every live range
  - Choose a live range to spill if necessary
  - If possible: remove a node from the graph - if the number of neighbors < k
- Select the node as a colorable node and push it to stack

Otherwise: choose a node to spill, remove the node and its incident edges. But still push the node into stack.

Insert spills and rebuild interference graph
CHAITIN-BRIGGS IN PRACTICE

2 Registers

Stack

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

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

2 Registers

Stack

Colors:
1: ⬅
2: ❄
Coalescing

• Coalescing happens during interference graph building
  * For any copy
    If LRx -> LRy, and LRx, LRy do not interfere, combine LRx & LRy into one live range and delete copy operation
  * Update interference graph I
    LRxy interferes with all of LRx and LRy's conflicts.

• Coalescing is a circular problem
  * Reduces the number of live ranges -> determines the cost of building graph
  * Coalescing needs a graph to know if two live ranges interfere
  * Need to iterate build-coalesce process

Engineering a Compiler. Chapter 13.4.6
Conservative Coalescing

• Chaitin’s scheme coalesces **every copy** that it can
  * Coalescing i and j can recreate \( \text{deg}(ij) > \max(\text{deg}(i), \text{deg}(j)) \)
    — May make ij harder to color
  * We can limit coalescing to **conservative** cases
    — Idea: Only create ij if it will get a color
    — Tempting to say that we need \( \text{deg}(ij) < k \), so ij is trivially colored
    — In fact, we need ij has fewer than \( k \) neighbors of significant degree
  * We can also bias the color section (bias coalescing)
    — If i and j are connected by a copy, try to pick the same color
    — If the other node is not yet colored, pick a color still open for it
    — Can generalize to multiple copies (by only immediate neighbors)
    — Biased by chance, might need some lookahead
Iterated Coalescing

• Simplifying may increases the chance of coalescing
  * Iterate simplify and coalesce
    — Simplify non-move related nodes first
    — If no trivially colored non-move nodes remain, try to coalesce
    — Coalescing reduces degree in the graph and help simplify
    — Only using conservative coalescing

  * Invented for Standard ML of New Jersey
    — Long parameter lists, passed in registers
    — This adds many additional edges
    — Iterated coalescing cured their problem
Spilling Partial Live Ranges

- Splitting the live range
  * Split a live range into multiple live ranges so that there will be less interference in the interference graph making it k-colorable
  * Spill the value to memory and load it back at the points where the web is split instead of where it is used
  * Fits nicely into the “insert spill” phase in Chaitin style allocator
Live Range Splitting Example

Two colorable?
No!
Live Range Splitting Example

z = ...
    ... = z

x = ...
y = ...
    ... = x
    ... = x
    ... = y

.....
    ... = z

Two colorable?
Yes!
Live Range Splitting Example

Two colorable? Yes!
Iterated Coalescing

• **Bottom-up splitting (Chow)**
  * Break troubled live range into basic blocks
  * Recombine them when it does not increase degree

• **Agressive splitting (Briggs)**
  * Split aggressively based on the CFG (find strongly connected components)
  * Undo non-productive splits

• **Interference region splitting (Bergner)**
  * Spill just region that conflicts with colored nodes
  * Run in competition with default spilling

• **Passive splitting (Simpson)**
  * Use directed interference graph to identify splits
  * Run in competition with default spilling
Interference Region Splitting

• **Basic idea**
  * Find regions where i & j are both live
  * Spill i around this interference region (IR)
  * Can reduce total cost of spilling
  * Fits entirely in “Insert Spills” phase of a Briggs allocator

• **The implementation**
  * Take colored subgraph and list of uncolored nodes
  * For each uncolored LR, find subranges that can be colored
    — Rest of LR is IR
  * Compare cost of spilling IR versus cost of spilling entire LR
    — Take cheaper alternative
Passive Splitting

See Eric’s paper presentation
Bibliography

• Briggs, Cooper, & Torczon, “Improvements to Graph Coloring Register Allocation,” ACM TOPLAS 16(3), May, 1994.
• Cooper, Harvey, & Torczon, “How to Build an Interference Graph,” Software–Practice and Experience, 28(4), April, 1998