

CS738: Advanced Compiler Optimizations

Sparse Conditional Constant Propagation

Amey Karkare

karkare@cse.iitk.ac.in

<http://www.cse.iitk.ac.in/~karkare/cs738>

Department of CSE, IIT Kanpur



Sparse Simple Constant Propagation (SSC)

- ▶ Improved analysis time over Simple Constant Propagation

Sparse Simple Constant Propagation (SSC)

- ▶ Improved analysis time over Simple Constant Propagation
- ▶ Finds all simple constant

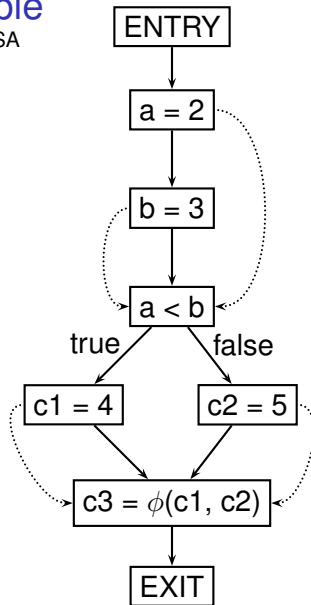
Sparse Simple Constant Propagation (SSC)

- ▶ Improved analysis time over Simple Constant Propagation
- ▶ Finds all simple constant
 - ▶ Same class as Simple Constant Propagation

Motivating Example

Dashed edges denote SSA

def-use chains



Preparations for SSC Analysis

- ▶ Convert the program to SSA form

Preparations for SSC Analysis

- ▶ Convert the program to SSA form
- ▶ One statement per basic block

Preparations for SSC Analysis

- ▶ Convert the program to SSA form
- ▶ One statement per basic block
- ▶ Add connections called *SSA edges*

Preparations for SSC Analysis

- ▶ Convert the program to SSA form
- ▶ One statement per basic block
- ▶ Add connections called *SSA edges*
 - ▶ Connect (unique) definition point of a variable to its use points

Preparations for SSC Analysis

- ▶ Convert the program to SSA form
- ▶ One statement per basic block
- ▶ Add connections called *SSA edges*
 - ▶ Connect (unique) definition point of a variable to its use points
 - ▶ Same as *def-use* chains

SSC Algorithm: Initialization

- ▶ Evaluate expressions involving constants only and assign the value (c) to variable on LHS

SSC Algorithm: Initialization

- ▶ Evaluate expressions involving constants only and assign the value (c) to variable on LHS
- ▶ If expression can not be evaluated at compile time, assign \perp

SSC Algorithm: Initialization

- ▶ Evaluate expressions involving constants only and assign the value (c) to variable on LHS
- ▶ If expression can not be evaluated at compile time, assign \perp
- ▶ Else (for expression contains variables) assign \top

SSC Algorithm: Initialization

- ▶ Evaluate expressions involving constants only and assign the value (c) to variable on LHS
- ▶ If expression can not be evaluated at compile time, assign \perp
- ▶ Else (for expression contains variables) assign \top
- ▶ Initialize worklist WL with SSA edges whose def is not \top

SSC Algorithm: Initialization

- ▶ Evaluate expressions involving constants only and assign the value (c) to variable on LHS
- ▶ If expression can not be evaluated at compile time, assign \perp
- ▶ Else (for expression contains variables) assign \top
- ▶ Initialize worklist WL with SSA edges whose def is not \top
- ▶ Algorithm terminates when WL is empty

SSC Algorithm: Iterative Actions

- ▶ Take an SSA edge E out of WL

SSC Algorithm: Iterative Actions

- ▶ Take an SSA edge E out of WL
- ▶ Take meet of the value at def end and the use end of E for the variable defined at def end

SSC Algorithm: Iterative Actions

- ▶ Take an SSA edge E out of WL
- ▶ Take meet of the value at def end and the use end of E for the variable defined at def end
- ▶ If the meet value is different from use value, replace the use by the meet

SSC Algorithm: Iterative Actions

- ▶ Take an SSA edge E out of WL
- ▶ Take meet of the value at def end and the use end of E for the variable defined at def end
- ▶ If the meet value is different from use value, replace the use by the meet
- ▶ Recompute the def d at the use end of E

SSC Algorithm: Iterative Actions

- ▶ Take an SSA edge E out of WL
- ▶ Take meet of the value at def end and the use end of E for the variable defined at def end
- ▶ If the meet value is different from use value, replace the use by the meet
- ▶ Recompute the def d at the use end of E
- ▶ If the recomputed value is *lower* than the stored value, add all SSA edges originating at d

Meet for ϕ -function

$$v = \phi(v_1, v_2, \dots, v_k)$$

$$\Rightarrow \text{ValueOf}(v) = v_1 \wedge v_2 \wedge \dots \wedge v_n$$

SSC Algorithm: Complexity

- ▶ Height of CP lattice = 2

SSC Algorithm: Complexity

- ▶ Height of CP lattice = 2
- ▶ Each SSA edge is examined at most twice, for each lowering

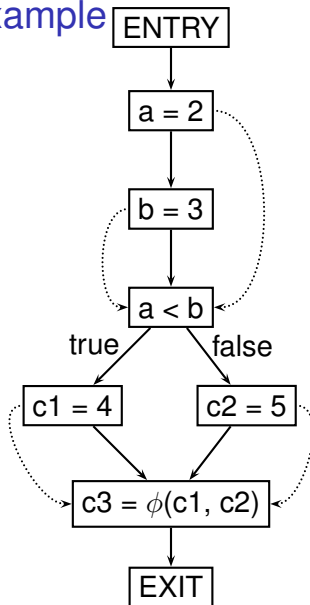
SSC Algorithm: Complexity

- ▶ Height of CP lattice = 2
- ▶ Each SSA edge is examined at most twice, for each lowering
- ▶ Theoretical size of SSA graph: $O(V \times E)$

SSC Algorithm: Complexity

- ▶ Height of CP lattice = 2
- ▶ Each SSA edge is examined at most twice, for each lowering
- ▶ Theoretical size of SSA graph: $O(V \times E)$
- ▶ Practical size: linear in the program size

SSC: Practice Example



SSC: Practice Example

What if we change “ $c1 = 4$ ” to “ $c1 = 5$ ”?

Sparse Conditional Constant Propagation (SCC)

- ▶ Constant Propagation with *unreachable code elimination*

Sparse Conditional Constant Propagation (SCC)

- ▶ Constant Propagation with *unreachable code elimination*
- ▶ Ignore definitions that reach a use via a non-executable edge

SCC Algorithm: Key Idea

$$v = \phi(v_1, v_2, \dots, v_k)$$

$$\Rightarrow \text{ValueOf}(v) = \bigwedge_{i \in \text{ExecutablePath}} v_i$$

We ignore paths that are not “yet” marked executable

SCC Algorithm: Preparations

- ▶ Two Worklists

SCC Algorithm: Preparations

- ▶ Two Worklists
 - ▶ Flow Worklist (*FWL*)

SCC Algorithm: Preparations

- ▶ Two Worklists
 - ▶ Flow Worklist (*FWL*)
 - ▶ Worklist of flow graph edges

SCC Algorithm: Preparations

- ▶ Two Worklists
 - ▶ Flow Worklist (*FWL*)
 - ▶ Worklist of flow graph edges
 - ▶ SSA Worklist (*SWL*)

SCC Algorithm: Preparations

- ▶ Two Worklists
 - ▶ Flow Worklist (*FWL*)
 - ▶ Worklist of flow graph edges
 - ▶ SSA Worklist (*SWL*)
 - ▶ Worklist of SSA graph edges

SCC Algorithm: Preparations

- ▶ Two Worklists
 - ▶ Flow Worklist (*FWL*)
 - ▶ Worklist of flow graph edges
 - ▶ SSA Worklist (*SWL*)
 - ▶ Worklist of SSA graph edges
- ▶ Execution Halts when **both** worklists are empty

SCC Algorithm: Preparations

- ▶ Two Worklists
 - ▶ Flow Worklist (*FWL*)
 - ▶ Worklist of flow graph edges
 - ▶ SSA Worklist (*SWL*)
 - ▶ Worklist of SSA graph edges
- ▶ Execution Halts when **both** worklists are empty
- ▶ Associate a flag, the *ExecutableFlag*, with every flow graph edge to control the evaluation of ϕ -function in the destination node

SCC Algorithm: Initialization

- ▶ Initialize *FWL* to contain edges leaving ENTRY node

SCC Algorithm: Initialization

- ▶ Initialize *FWL* to contain edges leaving ENTRY node
- ▶ Initialize *SWL* to empty

SCC Algorithm: Initialization

- ▶ Initialize *FWL* to contain edges leaving ENTRY node
- ▶ Initialize *SWL* to empty
- ▶ Each *ExecutableFlag* is false initially

SCC Algorithm: Initialization

- ▶ Initialize *FWL* to contain edges leaving ENTRY node
- ▶ Initialize *SWL* to empty
- ▶ Each *ExecutableFlag* is false initially
- ▶ Each value is \top initially (Optimistic)

SCC Algorithm: Iterations

- ▶ Remove an item from either worklist

SCC Algorithm: Iterations

- ▶ Remove an item from either worklist
- ▶ process the item (described next)

SCC Algorithm: Processing *FWL* Item

- ▶ Item is flow graph edge

SCC Algorithm: Processing *FWL* Item

- ▶ Item is flow graph edge
- ▶ If *ExecutableFlag* is true, do nothing

SCC Algorithm: Processing *FWL* Item

- ▶ Item is flow graph edge
- ▶ If *ExecutableFlag* is true, do nothing
- ▶ Otherwise

SCC Algorithm: Processing *FWL* Item

- ▶ Item is flow graph edge
- ▶ If *ExecutableFlag* is true, do nothing
- ▶ Otherwise
 - ▶ Mark the *ExecutableFlag* as true

SCC Algorithm: Processing *FWL* Item

- ▶ Item is flow graph edge
- ▶ If *ExecutableFlag* is true, do nothing
- ▶ Otherwise
 - ▶ Mark the *ExecutableFlag* as true
 - ▶ **Visit- ϕ** for all ϕ -functions in the destination

SCC Algorithm: Processing *FWL* Item

- ▶ Item is flow graph edge
- ▶ If *ExecutableFlag* is true, do nothing
- ▶ Otherwise
 - ▶ Mark the *ExecutableFlag* as true
 - ▶ **Visit**- ϕ for all ϕ -functions in the destination
 - ▶ If only one of the *ExecutableFlags* of incoming flow graph edges for dest is true (dest visited for the first time), then **VisitExpression** for all expressions in dest

SCC Algorithm: Processing *FWL* Item

- ▶ Item is flow graph edge
- ▶ If *ExecutableFlag* is true, do nothing
- ▶ Otherwise
 - ▶ Mark the *ExecutableFlag* as true
 - ▶ **Visit- ϕ** for all ϕ -functions in the destination
 - ▶ If only one of the *ExecutableFlags* of incoming flow graph edges for dest is true (dest visited for the first time), then **VisitExpression** for all expressions in dest
 - ▶ If the dest contains only one outgoing flow graph edge, add that edge to *FWL*

SCC Algorithm: Processing *SWL* Item

- ▶ Item is SSA edge

SCC Algorithm: Processing *SWL* Item

- ▶ Item is SSA edge
- ▶ If dest is a ϕ -function, **Visit- ϕ**

SCC Algorithm: Processing *SWL* Item

- ▶ Item is SSA edge
- ▶ If dest is a ϕ -function, **Visit- ϕ**
- ▶ If dest is an expression and any of *ExecutableFlags* for the incoming flow graph edges of dest is true, perform **VisitExpression**

SCC Algorithm: Visit- ϕ

$$v = \phi(v_1, v_2, \dots, v_k)$$

- ▶ If i^{th} incoming edge's *ExecutableFlag* is true, $val_i = \text{ValueOf}(v_i)$ else $val_i = \top$

SCC Algorithm: Visit- ϕ

$$v = \phi(v_1, v_2, \dots, v_k)$$

- ▶ If i^{th} incoming edge's *ExecutableFlag* is true, $val_i = \text{ValueOf}(v_i)$ else $val_i = \top$
- ▶ $\text{ValueOf}(v) = \bigwedge_i val_i$

SCC Algorithm: VisitExpression

- ▶ Evaluate the expression using values of operands and rules for operators

SCC Algorithm: VisitExpression

- ▶ Evaluate the expression using values of operands and rules for operators
- ▶ If the result is same as old, nothing to do

SCC Algorithm: VisitExpression

- ▶ Evaluate the expression using values of operands and rules for operators
- ▶ If the result is same as old, nothing to do
- ▶ Otherwise

SCC Algorithm: VisitExpression

- ▶ Evaluate the expression using values of operands and rules for operators
- ▶ If the result is same as old, nothing to do
- ▶ Otherwise
 - ▶ If the expression is part of assignment, add all outgoing SSA edges to *SWL*

SCC Algorithm: VisitExpression

- ▶ Evaluate the expression using values of operands and rules for operators
- ▶ If the result is same as old, nothing to do
- ▶ Otherwise
 - ▶ If the expression is part of assignment, add all outgoing SSA edges to *SWL*
 - ▶ if the expression controls a conditional branch, then

SCC Algorithm: VisitExpression

- ▶ Evaluate the expression using values of operands and rules for operators
- ▶ If the result is same as old, nothing to do
- ▶ Otherwise
 - ▶ If the expression is part of assignment, add all outgoing SSA edges to *SWL*
 - ▶ if the expression controls a conditional branch, then
 - ▶ if the result is \perp , add all outgoing flow edges to *FWL*

SCC Algorithm: VisitExpression

- ▶ Evaluate the expression using values of operands and rules for operators
- ▶ If the result is same as old, nothing to do
- ▶ Otherwise
 - ▶ If the expression is part of assignment, add all outgoing SSA edges to *SWL*
 - ▶ if the expression controls a conditional branch, then
 - ▶ if the result is \perp , add all outgoing flow edges to *FWL*
 - ▶ if the value is constant c , only the corresponding flow graph edge is added to *FWL*

SCC Algorithm: VisitExpression

- ▶ Evaluate the expression using values of operands and rules for operators
- ▶ If the result is same as old, nothing to do
- ▶ Otherwise
 - ▶ If the expression is part of assignment, add all outgoing SSA edges to *SWL*
 - ▶ if the expression controls a conditional branch, then
 - ▶ if the result is \perp , add all outgoing flow edges to *FWL*
 - ▶ if the value is constant c , only the corresponding flow graph edge is added to *FWL*
 - ▶ Value can not be \top (why?)

SCC Algorithm: Complexity

- ▶ Each SSA edge is examined twice

SCC Algorithm: Complexity

- ▶ Each SSA edge is examined twice
- ▶ Flow graph nodes are visited once for every incoming edge

SCC Algorithm: Complexity

- ▶ Each SSA edge is examined twice
- ▶ Flow graph nodes are visited once for every incoming edge
- ▶ Complexity = $O(\# \text{ of SSA edges} + \# \text{ of flow graph edges})$

SCC Algorithm: Correctness and Precision

- ▶ SCC is conservative

SCC Algorithm: Correctness and Precision

- ▶ SCC is conservative
 - ▶ Never labels a variable value as a constant

SCC Algorithm: Correctness and Precision

- ▶ SCC is conservative
 - ▶ Never labels a variable value as a constant
- ▶ SCC is at least as powerful as Conditional Constant Propagation (CC)

SCC Algorithm: Correctness and Precision

- ▶ SCC is conservative
 - ▶ Never labels a variable value as a constant
- ▶ SCC is at least as powerful as Conditional Constant Propagation (CC)
 - ▶ Finds all constants as CC does

SCC Algorithm: Correctness and Precision

- ▶ SCC is conservative
 - ▶ Never labels a variable value as a constant
- ▶ SCC is at least as powerful as Conditional Constant Propagation (CC)
 - ▶ Finds all constants as CC does
- ▶ PROOFs: In paper **Constant propagation with conditional branches** by **Mark N. Wegman, F. Kenneth Zadeck**, ACM TOPLAS 1991.

Practice Example

