A Sparse Algorithm for Predicated Global Value Numbering

Karthik Gargi

Hewlett-Packard India Software Operation

PLDI'02
Monday 17 June 2002

Sparse Predicated Global Value Numbering

1. Introduction
2. Brute Force Algorithm
3. Sparse Value Numbering
4. Additional Analyses and Balanced Value Numbering
5. Putting it all Together
6. Measurements
7. Conclusions

SSA Optimization Framework

Routine IR
↓
Translate IR into SSA form
↓
...
↓
Perform global value numbering
↓
Transform IR based on results of GVN
↓
...
↓
Translate IR out of SSA form
↓
Optimized IR

Global Value Numbering

- A *value* is a constant or an SSA variable
- Values can be partitioned into *congruence classes*
- Congruent values are identical for any possible execution of a routine
- Every congruence class has a representative value called a *leader*

Global Value Numbering (continued)

- Analysis phase - does not modify IR
- Inputs
  - Routine in SSA form
- Outputs
  - Congruence classes of routine
  - Values in every congruence class
  - Leader of every congruence class
  - Congruence class of every value

Global Value Numbering (continued)

- GVN can be unified with:
  - Constant folding
  - Algebraic simplification
  - Unreachable code elimination
- The results of GVN are used to perform:
  - Unreachable code elimination
  - Constant propagation
  - Copy propagation
  - Redundancy elimination

Brute Force Algorithm

1. Make all SSA variables have the value $\top$
2. Clear hash table to map expressions to values
3. For all instructions V $\leftarrow$ X *op* Y in RPO:

   Let E be the expression: Value-of(X) *op* Value-of(Y)

   Perform a hash table lookup on E:

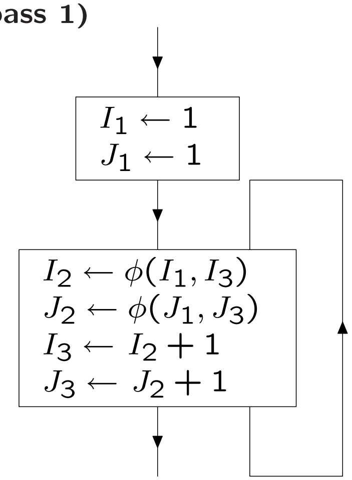
   - If lookup is successful, make its result the value of V
   - Otherwise set the value of V to V itself, and update hash table to map E onto V
4. Repeat steps 2. and 3. until there are no more changes in values

Brute Force Algorithm (example, pass 1)

Var	Value
I_1	$\top$
J_1	$\top$
I_2	$\top$
J_2	$\top$
I_3	$\top$
J_3	$\top$

Var	Value
I_1	I_1
J_1	I_1
I_2	I_1
J_2	I_1
I_3	I_3
J_3	I_3

Expr	Value
1	$\longrightarrow I_1$
$I_1 + 1$	$\longrightarrow I_3$

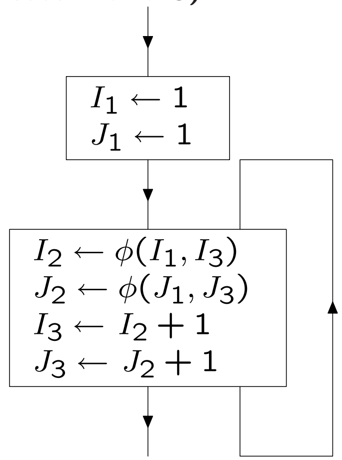


Brute Force Algorithm (example, passes 2 and 3)

Var	Value
I_1	I_1
J_1	I_1
I_2	I_1
J_2	I_1
I_3	I_3
J_3	I_3

Var	Value
I_1	I_1
J_1	I_1
I_2	I_2
J_2	I_2
I_3	I_3
J_3	I_3

Expr	Value
$1 \longrightarrow$	I_1
$phi(I_1, I_3) \longrightarrow$	I_2
$I_2 + 1 \longrightarrow$	I_3



Brute Force Algorithm (continued)

- This is Taylor Simpson's hash based RPO algorithm (1996)
- Achieves the same result as partitioning algorithm of Alpern, Wegman and Zadeck (1988)
- Makes the *optimistic* assumption - all values are initially congruent, until proven otherwise
- Only an optimistic algorithm can discover the congruence of I_3 and J_3 in the previous example
- Takes $O(C)$ passes where C is the loop connectedness of the SSA def-use graph

Sparse Value Numbering

1. Initialize as in Brute Force
2. Touch instructions of start block
3. For all instructions I in RPO:

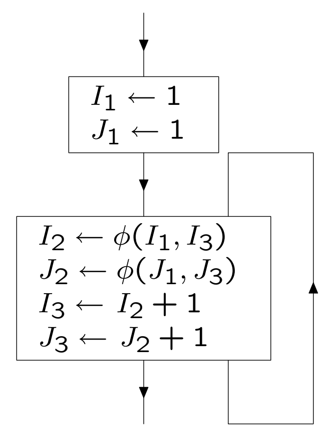
   If I is touched:

   - Wipe it
   - Process it as in Brute Force
   - If its value has changed, touch its consumers (found from SSA def-use chains)

4. Repeat step 3. until there are no more touched instructions

Sparse Value Numbering (continued)

- After every pass, values are the same as for Brute Force
- First pass processes 6 instructions, and leaves the definitions of I_2 and J_2 touched
- Second pass processes 4 instructions, and leaves the definitions of I_2 and J_2 touched
- Third pass processes 2 instructions, and confirms fixed point
- ≈1.5X faster than Brute Force for this example



Sparse Value Numbering (continued)

- Faster than Brute Force because it does not process all instructions in every pass
- Has to examine every instruction to check if it is touched, but this is much faster than processing it
- Does not clear hash table between passes
- When the leader of a congruence class is moved to a new congruence class:
  - Touch the definitions of the remaining members of the old class
  - Choose one of them to be the new leader of the old class

Sparse Value Numbering (continued)

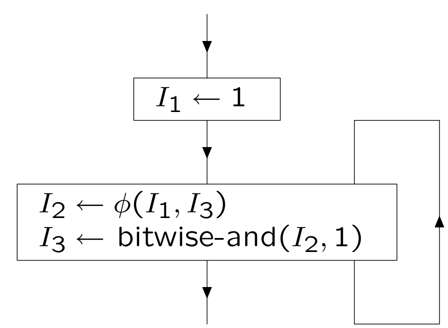
- For acyclic code, takes one pass
- For cyclic code, when the optimistic assumption is confirmed, takes almost one pass
- For cyclic code when the optimistic assumption is rejected, takes anywhere up to one less pass than Brute Force
- Measurements from SPEC CINT2000 C benchmarks:
  - Value numbering (unified with additional analyses) takes $< 4\%$ of total optimization time
  - 1.98 passes per routine on average
  - Speedup due to sparseness is 1.23–1.57

Algebraic Transformations

- Before looking up an expression in the hash table:
  - Perform constant folding
  - Perform algebraic simplification
  - Perform global reassociation
  - Apply distributive law
- If any value of a congruence class is defined to be a constant, make that constant the leader of the congruence class

Algebraic Transformations (continued)

- First pass sets value of I_1 to 1
- Ignoring I_3, value of I_2 is also 1
- Constant folding evaluates I_3 to 1
- Second pass processes definition of I_2
- The value of I_2 remains 1
- Hence I_3 has the value 1
- Almost one pass to reach fixed point

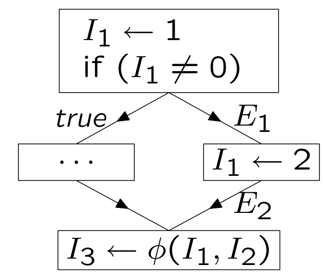


Unreachable Code Elimination

- Assume start block is initially reachable
- Assume all other blocks and edges are initially unreachable
- Wipe but do not process, touched but unreachable instructions
- Examine jump instructions also:
  - If an outedge cannot be followed, it remains unreachable
  - Otherwise it becomes and remains reachable
- Once an edge becomes reachable, so do its target blocks
- Ignore operands of ϕ-functions carried by unreachable edges.

Unreachable Code Elimination (continued)

- Constant folding evaluates the predicate $I_1 \neq 0$ to true
- So edges E_1 and E_2 remain unreachable
- So I_2 is ignored when evaluating the definition of I_3
- Hence I_3 has the value 1

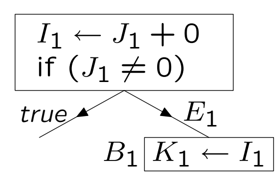


Balanced Value Numbering

- Pessimistic in congruence of values - assumes all values are non-congruent until proven otherwise
- Optimistic in reachability
- To perform balanced value numbering:
  - Treat every cyclic ϕ-function as a unique value
  - Terminate after the first pass
- On SPEC CINT2000 C benchmarks:
  - As fast as pessimistic value numbering
  - Almost as strong as optimistic value numbering
  - Runs 1.39–1.90 times faster than optimistic value numbering

Value Inference

- The use of I_1 in block B_1 is dominated by edge E_1
- The predicate $J_1 \neq 0$ has the value false at edge E_1
- So J_1 has the value 0 at edge E_1 and block B_1
- I_1 is congruent to J_1
- So I_1 has the value 0 at edge E_1 and block B_1
- Hence K_1 has the value 0



Value Inference (continued)

- Algorithm:

  Before looking up an expression in the hash table:

  For each operand X of the expression:

  1. Start from the block B containing the expression
  2. Go up the dominator tree looking for an edge E such that:
     (a) E dominates B
     (b) E is the true outedge from a jump instruction with predicate $Y = Z$
     (c) Y is congruent to X
  3. If such an E is found, then replace X by Z

- Only dominator tree based approach can be completely unified with value numbering

Value Inference (continued)

- Two ways to determine dominance relationships:
  - Complete algorithm - incrementally build reachable dominator tree
  - Practical algorithm - use dominator tree of routine
    * Cannot ignore unreachable code
    * Cannot perform inferences along back edges
- When the reachability or predicate of an edge $B_1 \rightarrow B_2$ changes, touch potentially affected instructions:
  - Complete algorithm - touch all instructions of blocks dominated by block B_2
  - Practical algorithm - touch all instructions downstream in RPO of block B_2

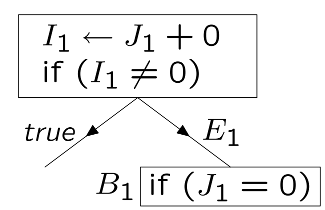
Value Inference (continued)

- Value inference can take $O(E^2)$ time in the worst case, where E is the number of edges in the CFG
- Sufficient to perform value inference on operands of $=$ or $\neq$ predicates of jump instructions
- Track the number of such values in every congruence class
- Perform value inference only on values in classes with positive counts
- Results of value inference can be cached across multiple uses in a block
- Measurements from SPEC CINT2000 C benchmarks:

  Value inference visits 0.91 blocks per instruction on average

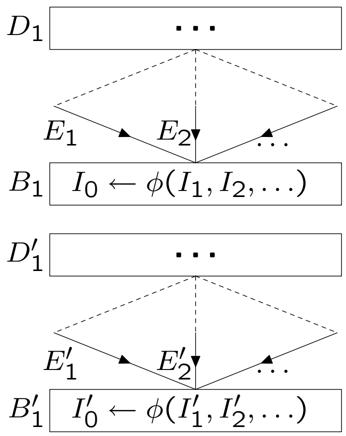
Predicate Inference

- Similar to value inference
- The predicate $J_1 = 0$ in block B_1 is dominated by edge E_1
- The predicate $I_1 \neq 0$ has the value false at edge E_1
- I_1 is congruent to J_1
- So the predicate $J_1 = 0$ has the value true in block B_1



Φ-Predication

- Problem: when are I_0 and I'_0 congruent?
- Rewrite I_0 as: if P_1 then I_1 else if P_2 then I_2 else if ...
- P_1 is true when and only when control reaches B_1 along $D_1 \rightarrow \cdots \rightarrow E_1 \rightarrow B_1$
- Similarly I'_0 is: if P'_1 then I'_1 else if P'_2 then I'_2 else if ...
- I_0 is congruent to I'_0 if I_i is congruent to I'_i and P_j is congruent to P'_j

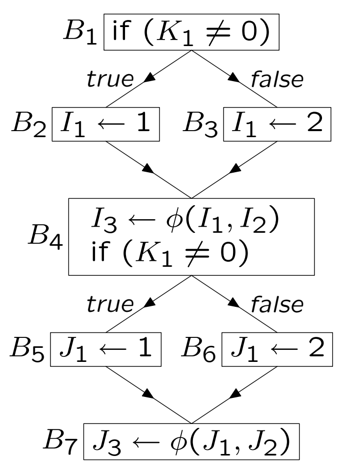


Φ-Predication (continued)

- Predicate of block B_1 defined as: $P_1 \vee P_2 \vee \ldots$
- Two ϕ-functions are congruent if their arguments are congruent and either their blocks are identical or the predicates of their blocks are congruent
- To compute the predicate of block B_1:
  - Find its immediate dominator D_1
  - Traverse all reachable paths from block D_1 to block B_1
  - Combine predicates of jumps encountered during traversal
- Restrictions:
  - Block B_1 must postdominate block D_1
  - Back edges can not be traversed

Φ-Predication (continued)

- To determine the predicate of block B_4, start from block B_1
- Traverse the paths $B_1 \rightarrow B_2 \rightarrow B_4$ and $B_1 \rightarrow B_3 \rightarrow B_4$
- The predicate of block B_4 is: $(K_1 \neq 0) \vee (K_1 = 0)$
- The predicate of block B_7 is identical
- Hence J_3 is congruent to I_3



Φ-Predication (continued)

- Compute predicates of touched blocks only
- Compute predicate of block before processing instructions of block
- When the reachability or predicate of an edge $B_1 \rightarrow B_2$ changes, touch potentially affected blocks:
  - Complete algorithm - touch all blocks that postdominate block B_2
  - Practical algorithm - touch all blocks downstream in RPO of block B_2
- Measurements from SPEC CINT2000 C benchmarks:

  Φ-predication visits 0.16 blocks per instruction on average

Putting it all Together

- Unifies sparse value numbering with constant folding, algebraic simplification, unreachable code elimination, global reassociation, value inference, predicate inference, and ϕ-predication
- Worst case time complexity:
  - Balanced value numbering - $O(E^2(E+I))$
  - Optimistic value numbering:
    - Acyclic CFG - $O(E^2(E+I))$
    - Cyclic CFG - $O(CE^2(E+I))$
- Measurements from SPEC CINT2000 C benchmarks:

  Unified algorithm takes $<4\%$ of total optimization time

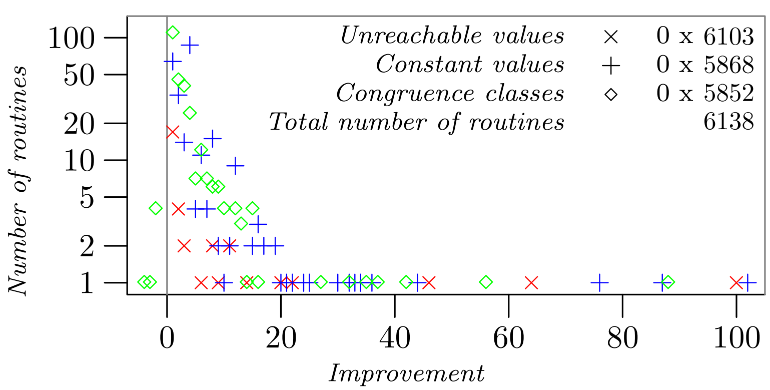
Measurements

Unified algorithm on SPEC CINT2000 C benchmarks:

- Value numbering (unified with additional analyses) takes $<4\%$ of total optimization time
- Runs 1.23–1.57 times faster when sparseness is enabled
- Runs 1.15–1.32 times faster when global reassociation, value inference, predicate inference and ϕ-predication are disabled
- Runs 1.39–1.90 times faster with balanced value numbering
- 1.98 passes per routine on average
- Blocks visited per instruction on average:
  Value inference - 0.91
  Predicate inference - 0.38
  Φ-predication - 0.16

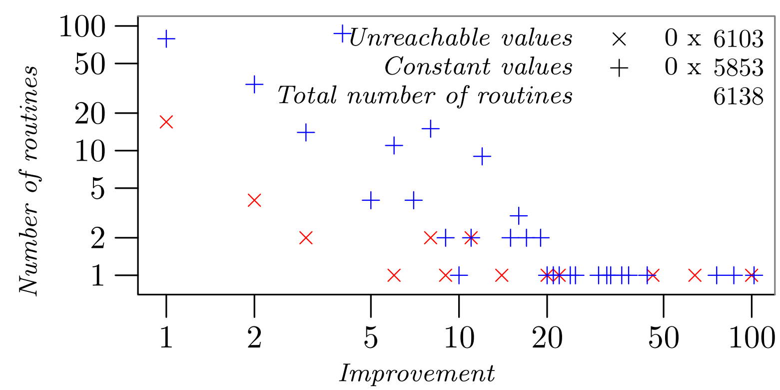
Measurements (continued)

Unified algorithm vs. Click's strongest algorithm (1995) on SPEC CINT2000 C benchmarks:



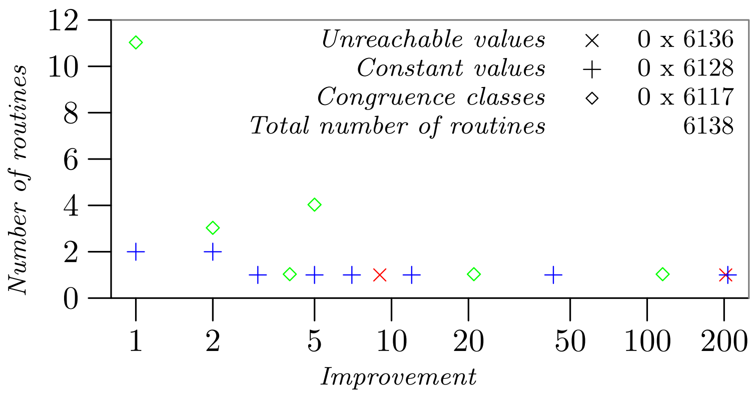
Measurements (continued)

Unified algorithm vs. Wegman and Zadeck's sparse conditional constant propagation algorithm:



Measurements (continued)

Unified algorithm: optimistic vs. balanced value numbering



Conclusions

- Sparse value numbering is practical and efficient
- Balanced value numbering is a good tradeoff between compilation time and optimization strength
- Sparse value numbering can be unified with a wide range of additional analyses
- The unified algorithm offers modest improvements over existing methods

Thank you to Laurie Hendren for helping to prepare and presenting this slide set.

Questions or comments regarding this work may please be sent to the author at kg@india.hp.com

Examples of Differences - Unreachable Values (Unified vs. Click)

- Benchmark: 176.gcc
- Routine: try_combine (instruction combiner)
- Unreachable values:
  - Click 95: 0
  - Unified algorithm: 100
  - Improvement: 100
- Source: Predicate inference

```
if (X != 0)
   ...
   if (X != 0)
     ...
   else
     ...
   ...
if (X < 64)
  ...
  if (X >= 64)
    ...
```