An Efficient Static Analysis Algorithm to Detect Redundant Memory Operations

Keith D. Cooper & Li Xu
Department of Computer Science
Rice University
Houston, Texas, USA

keith@rice.edu

The Big Picture

Compiled code runs modern systems well below peak speeds

- Loads and stores take lots of time
  - Memory is farther away
  - Complex hierarchy makes costs depend on context
- Lots of work has been done on analysis
  - Myriad papers on points-to & alias analysis
  - Range of costs and precisions

In this work, we focused on ways to eliminate loads & stores

- Started as a study of the Media Bench codes
- Began by staring at lots of compiled code

Approaches to the Problem

We considered three distinct approaches

- Framework(s) based on partial redundancy elimination
  - Extend & improve them (*maybe using SSA form*)
  - Limited to lexical identity (*with help from SSA*)
- Ad hoc transformations
  - Has been done with some success
  - We had no new ideas here
- Value numbering
  - Unified framework
  - Try to track values through RAM

Strong notion of identity
Works well with our IR
Major push from advisor

Context

MSCP Research Compiler Infrastructure

- Suite of well-established optimizations
- Graph-coloring register allocator
- Back ends for SimpleScalar and for simulating IR

IR is a critical issue

- Low-level, three-address code
- Front-end tags each memory op & call site
  - Tags are invented names related to source-level names
  - Direct & concrete representation of ambiguity

What did we do?

- Extend Simpson's SCCVN to track values through RAM
  - Optimistic, global value numbering on SSA form
  - Find statically-detectable, (fully) redundant memory ops
- Built on tags + interprocedural pointer analysis
  - Provides needed base information
  - Wanted to solve problem *in our compiler*
- Added rules to handle memory-based values
  - Produced simple, easily understood scheme
  - Effective at improving code

Comparisons against same compiler without the new rules

Value Numbering

Basic idea is ancient & straightforward

Algorithm credited to Balke

Idea dates to Floyd, Ershov

- Assign unique number to each "value"
  - x & y have same number ⇔ x & y have same value
  - x & y have same number ⇒ remove computation of x or y

The Algorithm

For operation x ← y OP z in block

1. Get value numbers for y and z
2. Hash <vn(y),OP,vn(z)> to get vn(x)
3. vn(x) exists ⇒ replace op
4. y & z are constants ⇒ evaluate y OP z and fold result

Finds & replaces redundant operations

Folds local constants

Easily extended to handle algebraic identities

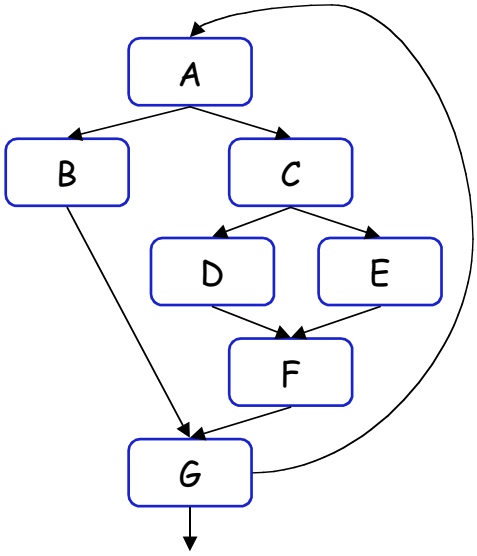
x*1 = x, y+0 = y, 2*z = z+z, …

Works on basic blocks *

Value Numbering Larger Regions

Extended Basic Blocks are easy

- Use table from predecessor as initial information



EBBS are AB, ACD, ACE, F, & G

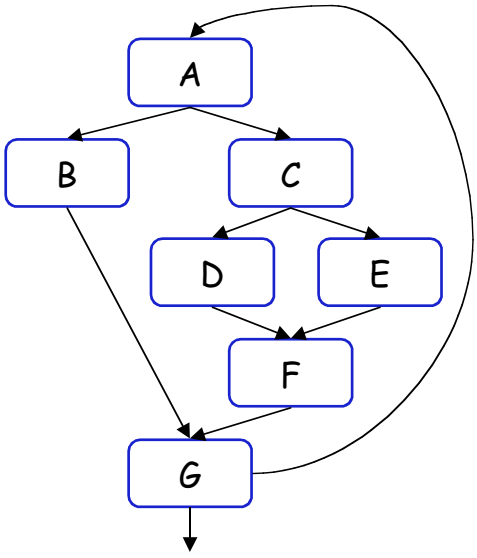
- Use A's table to start B & C
- Use C's table to start D & E
- F & G are on their own

Shift to SSA name space & use scoped symbol table to handle rollbacks

Value Numbering Larger Regions

Dominators let us handle even larger regions

- C dominates F and A dominates G



Dominator regions get more

- Use A's table to start B, C, & G
- Use C's table to start D, E, & F

Maximal acyclic regions

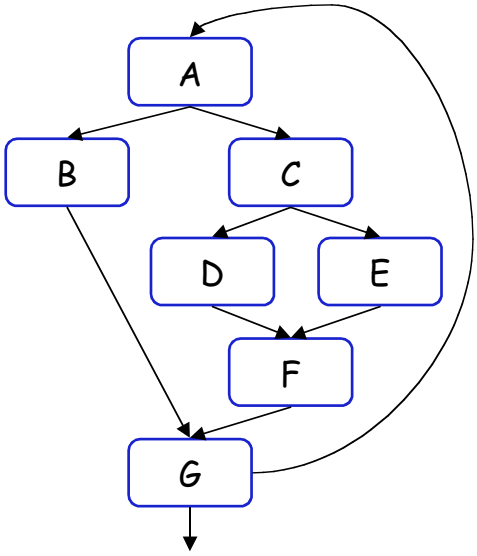
- Propagates along every forward edge in the CFG

What about the cycles?

Global Value Numbering

To handle cycles, iterate

- One table is pessimistic, two tables are optimistic



Two table version:

- HopedFor and Proven tables
- Iterate using HopedFor
- When stable, go back & copy facts into Proven table

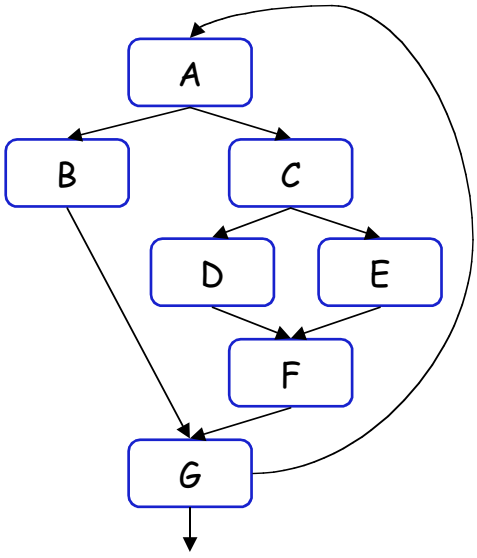
Fast algorithm

- Congruences of Alpern et al.
- Constants of Wegman-Zadeck sparse simple constant
- Algebraic identities

Global Value Numbering

One final complication

- Optimistic algorithm requires offline replacement



Several possible approaches:

- Dominance-based
- AVAIL-based
- LCM-based

Here, we used an AVAIL scheme

- Rename to reflect redundancy
- Compute AVAIL *(no kills)*
- Make a (local) rewriting pass

Tracking Memory Values

We added a new kind of tag, the Memory list (M-list)

- Produced by alias analysis

  Flow-insensitive, context-insensitive, Andersen-style pointer analysis consumes original tags & produces M-lists

- Two flavors: M-DEF & M-USE, with obvious meanings

M-lists are attached to select operations

FRAME : M-DEF has all objects the procedure might access

JSR : M-DEF is MOD, M-USE is REF

Loads : M-USE has all objects it might read

Stores : M-DEF has all objects it might write

Tracking Memory Values

Using M-lists

- SSA-construction : list entries become definitions & uses
  - In FRAME, they are all definitions (*initial state*)
  - In others, DEFs and USEs follow the names
  - Insert ϕ-functions for M-list entries
- Value numbering : new rules for each annotated operation
  - The rules fit into the classic algorithm
  - They reveal redundancy (*opportunity*)

We simply apply the old algorithm, with the new rules

Tracking Memory Values

The additional rules for M-lists

FRAME *size* ⇒ *argument list* M-DEF[n_0, n_1, …, n_i]

- For each $n \in$ M-DEF, set vn(hash(n)) to n
- We can use SSA names as value numbers (*They are unique*)

JSR *label*, r_0, r_1, …, r_i M-USE[n_0, n_1, …, n_j] M-DEF[m_0, m_1, …, m_k]

- For each $m \in$ M-DEF, set vn(hash(m)) to m
- The M-USE entries have no impact

Loads & stores are more complicated

Tracking Memory Values

The Additional Rules for M-lists

Need function to convert loads (& stores) to canonical form

LOAD *address* ⇒ *reg* M-USE[n_0, n_1, …, n_k]

- Hash <vn(address),LOAD> to get the vn — vn(OP_1)
- If vn(OP_1) exists, compare the value numbers of the M-USE entries of OP_1 against those of matching OP_2
  - Equality implies OP_1 is redundant
    - →Set vn(r_v) to the vn of OP_2's result
  - Inequality implies OP_1 is not provably redundant
    - →Set vn(r_v) to r_v and vn(OP_1) to r_v
    - →Store OP_1's M-DEF set with its vn table entry

Tracking Memory Values

The Additional Rules for M-lists

Store *reg* ⇒ *address* M-USE[n_0, n_1, …, n_j] M-DEF[m_0, m_1, …, m_k]

- Hash <vn(address),STORE> to get the vn — vn(OP_1)
- If vn(OP_1) exists, defined by OP_2 compare the value numbers of the M-USE entries of OP_1 against those of OP_2
  - Equality implies OP_1 is redundant
    - →Set vn(α) to vn(α') where $\alpha \in$ M-DEF_1, $\alpha' \in$ M-USE_2, and both are SSA names for the same memory object
  - Inequality implies OP_1 is not provably redundant
    - →Set vn(m_i) to m_i, for all $m_i \in$ M-DEF
    - → Create vn(<vn(*address*),STORE>) with value *reg* and store M-DEF_1 with that entry

How well does it work?

We implemented it in the MSCP framework

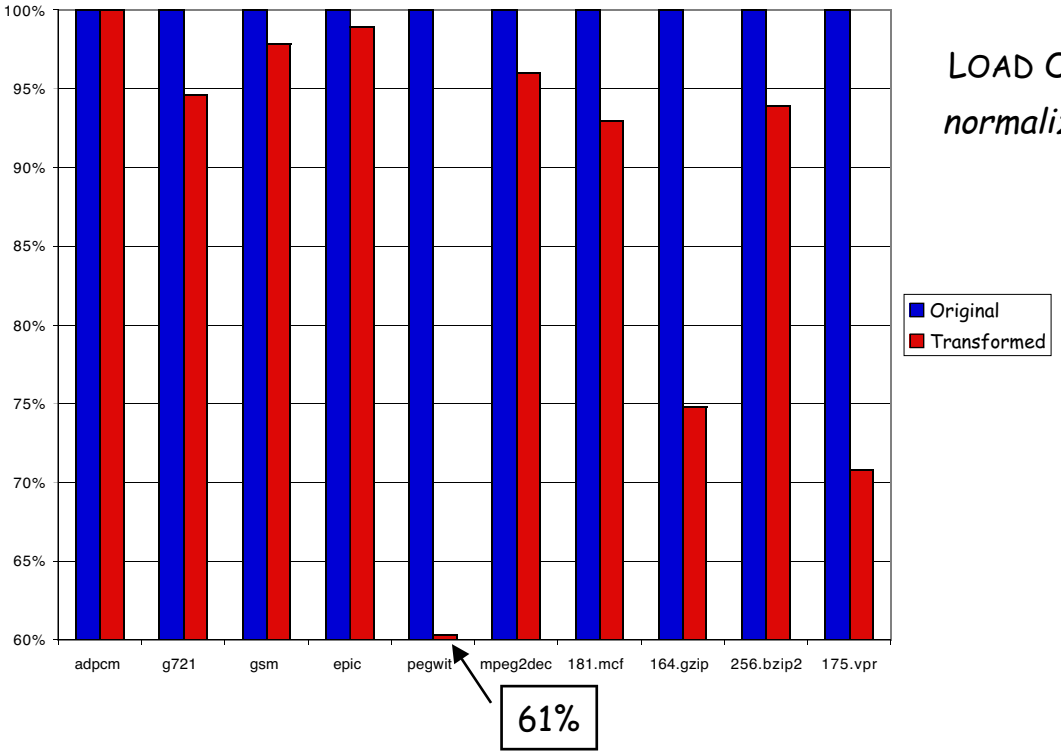
- Suite of other optimizations
- Chaitin-Briggs allocation for 32 register model
- Simulated the IR operations, measured operation counts
  - Uniform cost for operations

We compiled a set of 10 codes from MediaBench & Spec

- 59 to 5,610 basic blocks
- 1.6 Million to 10.8 Billion dynamic operations

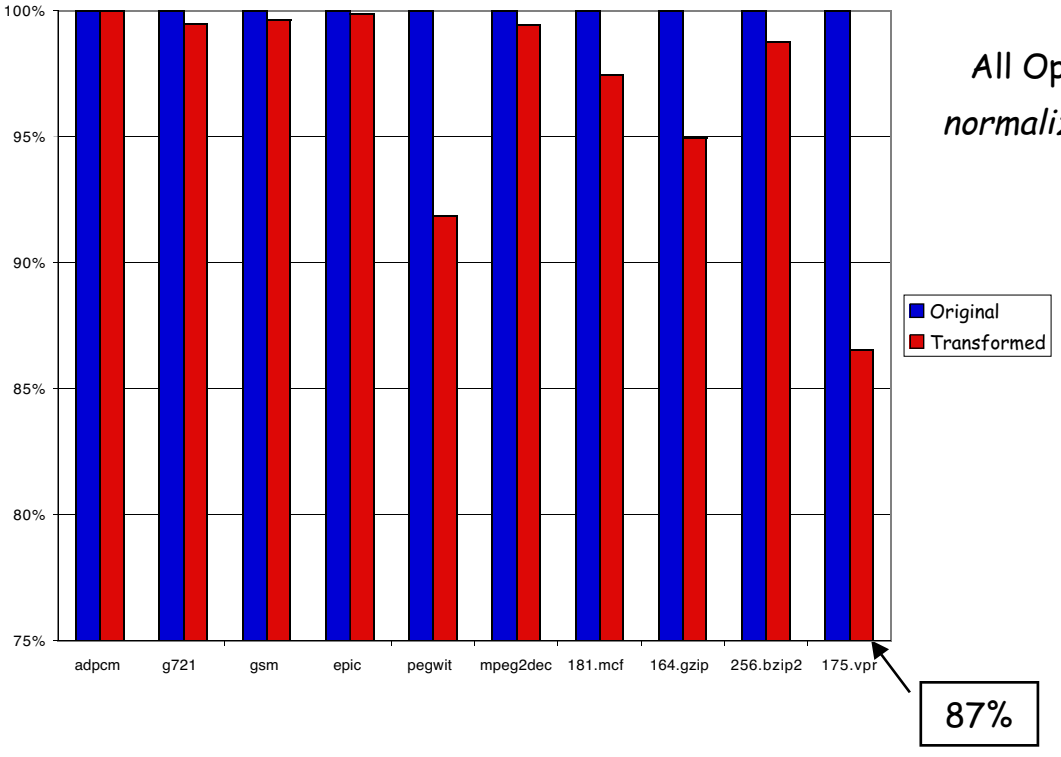
More statistics in the paper

How well does it work?



LOAD Ops
normalized

How well does it work?



All Ops
normalized

How well does it work?

What about stores?

- Eliminated six stores — dynamic count
  - Out of 10's of billions of operations
  - One double assignment, one in expanded macro
- Programmers don't write stores that it can eliminate

Did we expose new scalar redundancy?

- Sometimes yes (6/10), sometimes no (3/10)
  - Most cases made less than 1% difference in other ops
  - Best case was 4.3% improvement in other ops
- Can make things worse (1/10)
  - Epic lost by 23 out of 54,548,643 ops

Summary

- Fit results of pointer analysis into tags on the IR
  - M-USE and M-DEF sets on memory operations
  - Any analysis (almost) can be made to work
- Extended value numbering to track memory-based values
  - New rules for loads, stores, calls, & initialization
  - Will work in all the value-numbering scopes
- Found and eliminated some loads
  - 0 to 39% fewer loads, 0 to 13% fewer total operations
  - Statically-detectable, (fully) redundant loads & stores

Summary

Potential Improvements

- Use stronger analysis
  - Might find more redundancies
  - Studies in the literature are pessimistic
- Use better replacement technique
  - AVAIL-based replacement finds full redundancy
  - Using LCM-based redundancy might get more
    - → Might hurt code size, help execution time

Need more work on transformations for pointer-based codes

Results on a "real" architecture?

We also compiled the smaller codes for SimpleScalar

- Used single-issue model
- Could only simulate smaller codes (*Mediabench*)
- Results were similar to those reported here

SimpleScalar results led to erroneous comparisons

- Different parameters produce wildly different numbers
  - Sensitive to operation costs & scheduling algorithms
- Reports of other work use different parameters