Value numbering: Objectives

Four objectives

1. assign an identifying number to each variable / expression / constant:

   $$x \text{ \& } y \text{ have same number} \Leftrightarrow x = y \; \forall \text{ inputs}$$

2. use algebraic identities to simplify expressions
3. discover redundant computations and replace them
4. discover constant values, fold & propagate them

A Representation for Symbolic Expressions

- Value numbers can be used to represent symbolic expressions
- *Benefit*: Unique node for each value number
- Single basic block: Can use a DAG
- Multiple basic blocks: Directed graph

Value numbering: History

Long history in literature

- form of redundancy elimination: *cf.* CSE, PRE
- local version using hashing: late 60's Cocke & Schwartz, 1969
- algorithms for blocks, extended blocks, dominator regions, entire procedures, and (maybe) whole programs
- easy to understand algorithm for single block
- larger scopes $\Rightarrow$ more complex algorithms

References:

1. Alpern, Wegman & Zadeck, "Detecting Equality of Variables in Programs," *Proceedings POPL* 1988
2. Cooper & Simpson, "SCC-Based Value Numbering," *Rice University TR CRPC-TR95636-S*, 1995.
3. (For interest only) Briggs, Cooper, Simpson, "Value Numbering," *Software–Practice and Experience*, June 1997.

Local algorithm

Key notions

- each variable, expression, & constant gets a “value number”
  - same value number $\Rightarrow$ same value
  - equivalence based solely on facts from within block
- if an instruction’s value number is already defined, it can be eliminated & subsequent references subsumed
- constant folding is simple

Assumptions

1. low-level intermediate code
2. can find basic blocks (leader construction)

Local value numbering (cont'd)

Example

	a ← x + y	$V_1 \leftarrow$ hash(+,$VN[x]$,$VN[y]$), Name[V_1] ← a
	b ← x + y	*replace* with b ← a
	a ← 1	$V_a \leftarrow V_1$
⋆	c ← x + y	*replace how?*
⋆	d ← y + x	*replace how?*
	e ← d - 1	
⋆	f ← e + 1	*replace how?*

What's tricky?

- tracking where each value resides
- commutativity $\Rightarrow$???
- identities (e.g., V_x OR V_x, $V_y \times 1$): $\Rightarrow$ instr. gets vn of operand (V_x)

Local value numbering

The algorithm:

For each instruction i : x $\leftarrow$ y op z in the block

$V_1 \leftarrow$ VN[y] create if necessary

$V_2 \leftarrow$ VN[z] create if necessary

let v = $\texttt{hash}(op, V_1, V_2)$

if (v exists in hash table)

replace RHS with Name[v]

else

enter v in hash table

VN[x] $\leftarrow v$

Name[v] $\leftarrow$ t_i (new temporary)

replace instruction with: $t_i \leftarrow$ `y op z;` $x \leftarrow t_i$

Extending the basic hashing step

We can add many special cases

1. if all operands have the same value number
   a. if *op* is MAX, MIN, AND, OR, . . .
      replace *op* with a copy operation
   b. if *op* tests equality,
      replace it with a load immediate of true
   c. if *op* tests inequality
      replace it with a load immediate of false

2. if all operands are constants and we haven't already simplified the expression, then evaluate and propagate constants

Extending the basic hashing step (cont'd)

3. if one operand is constant and we haven't yet simplified the expression
   a. if constant operand is zero,
      I. replace ADD & OR with a copy
      II. replace MULT, DIV, AND with a LOADI
   b. if constant operand is one, simplify MULT into a copy

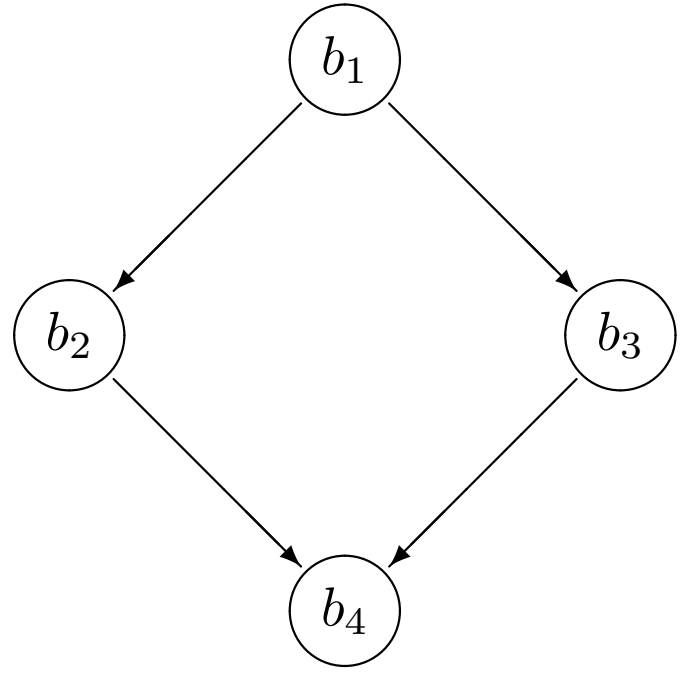
4. if *op* commutes, reorder its operands into ascending order by value number

Order of search space determines complexity

Extending to the Global Case

The Dominator-based VN Technique (DVNT)

- B_2, B_3 can be value-numbered using B_1's table
- So can B_4! $\Longrightarrow$
  At merge, use iDOM's table.
  Remove expressions killed in b_2, b_3
- Still based on hashing $\Longrightarrow$
  - algebraic identities
  - constant folding



Extending to the Global Case (cont'd)

Problems?

1. Dealing with merges:

   Why did our scheme fall apart in b_4?
   - A variable may be defined in B_2, B_3, or both
   - $\Rightarrow$ difficult to merge these tables
2. Equivalent expressions in loops.

This is not a truly global algorithm.
$\Rightarrow$ We need more powerful analysis tools

Examples Motivating A Global Approach

Hash-based approach is very effective:

```
// Assume:  X = Y
A ← X - Y
B ← Y - X
C ← A - B
D ← B - A
```

Global approach is needed:

```
X0 ← 1
Y0 ← 1
while (...) {
    X1 ← φ(X0, X2)
    Y1 ← φ(Y0, Y2)
    X2 ← X1 + 1
    Y2 ← Y1 + 1
}
```

A Global Approach (Alpern, Wegman & Zadeck)

Definitions

> *Instructions i & j are congruent iff they have the same operator and their operands are congruent*

Partitioning algorithm for SSA form

1. partition instructions into congruence classes by opcode
2. *worklist* $\leftarrow$ all classes
3. while (*worklist* is not empty)
   a. remove a class c from *worklist*
   b. for each class s that uses some $x \in c$
      while ($s \neq \phi$) do
      i. split s into s & s': all users of c in one class
      ii. put smaller of s & s' onto *worklist*
4. pick a representer for each partition & perform replacement

A Global Approach (Alpern, Wegman & Zadeck) (cont'd)

Miscellaneous observations

- based on Hopcroft-Ullman's DFA minimization *AS&U, 3.9*
- cannot prove $5 \times 2 \cong 7 + 3$ or $3 + 1 \cong 2 + 2$ or $x \times 1 \cong x$
- need separate pass to transform code (partitioning must complete first)
- powerful technique but ignores compile-time costs

Global value numbering *SCC/VDCM*

A,W, & Z partitioning algorithm

Strengths

- global scope
- optimism
- builds new names

Weaknesses

- $x \cong y \Rightarrow x + x \cong 2 \times y$
- constant folding
- compile-time speed

Global value numbering

- would like speed & flexibility of hash-based techniques
- would like optimism & global scope of partitioning technique

SCC/VDCM

- simple algorithm, based on Tarjan's **SCC**-finder
- replacement via Lazy Code Motion **(LCM)**

$\Rightarrow$ this should be the method of choice

Assumes code in SSA-graph form

BASIC RPO ALGORIHM

```
for all SSA names, i
    VN[i] ← ⊤

repeat
    done ← TRUE
    for all blocks b in reverse postorder (RPO)
        for all definitions x in b
            temp ← lookup(x.op, VN[x[1]], VN[x[2]], x)
            if (VN[x] ≠ temp)
                done ← FALSE
                VN[x] ← temp

    Remove all entries from the hash table
until done
```

The SCC-based Algorithm

Key Ideas

- Process one SCC at a time, in reverse depth-first order
- Use two hash tables:
  - Valid table only contains correct congruences
  - Optimistic table can contain incorrect congruences that are disproven later.
- Iterate over each SCC using Optimistic table
- Process different SCCs using Valid table: no iteration needed!
- *Don't* delete entries from Optimistic table after each SCC:
  Example: Optimistic entries from X SCC used in Y SCC to prove $Y_1 \cong X_1$.

The SCC-based Algorithm

initialize *optimistic* and *valid* tables for all nodes n
    n.*valnum* $\leftarrow \top$

while there is an unvisited node n
    DFS(n) (see previous lecture)

ProcessSCC(*SCC*)
    if *SCC* has a single member n
        Valnum(n, *valid*)
    else
        do
            changed $\leftarrow$ FALSE
            for each $n \in$ *SCC* in reverse postorder
                Valnum(n, *optimistic*)
        while *changed*
        for each $n \in$ *SCC* in reverse postorder
            Valnum(n, *valid*)

Tarjan's SCC algorithm drives the process.

Valnum(*node*, *table*)
    expr $\leftarrow$ $\langle$*node*$[1]$.*valnum*, *node*.*op*, *node*$[2]$.*valnum*$\rangle$
    Try to simplify *expr*
    temp $\leftarrow$ lookup(*expr*, *table*, *node*.*SSAname*)
    if *node*.*valnum* $\neq$ *temp*
        changed $\leftarrow$ TRUE
        node.*valnum* $\leftarrow$ *temp*

Wrap up on SCC

When process finishes

- $\rightarrow$ *x.valnum* $=$ *y.valnum* $\Leftrightarrow x \cong y$
- $\rightarrow$ *x.valnum* $= \top \Rightarrow$ *x* is uninitialized
- $\rightarrow$ constants have been folded
- $\rightarrow$ must rewrite code to reflect knowledge

Replacement

Technology of choice is Lazy Code Motion (Knoop *et al.*)

- avoids extraneous code motion
- has strong optimality proofs
- drawbacks: lexical identity & cost of solving equations

VDCM – an improvement to LCM

- encode value information into names (Briggs ‘94)

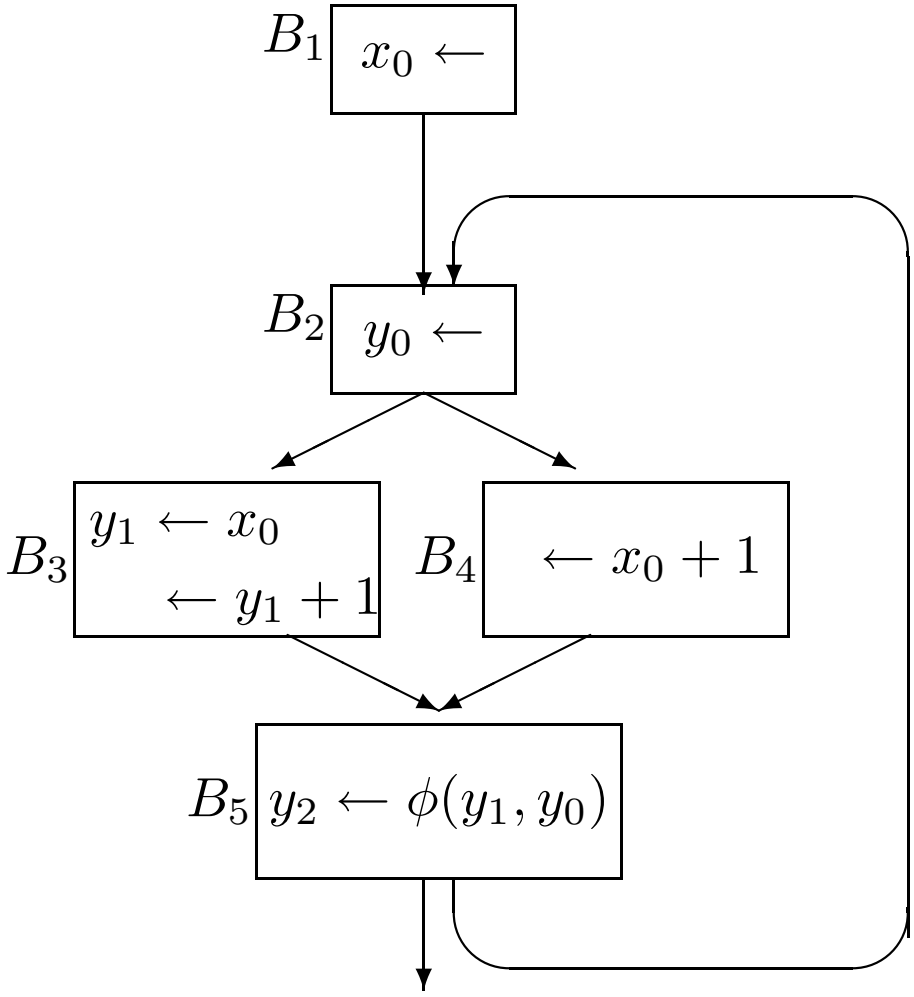
  $\rightarrow$ larger name space

- using SSA-names $\Rightarrow$ **KILL** set is empty

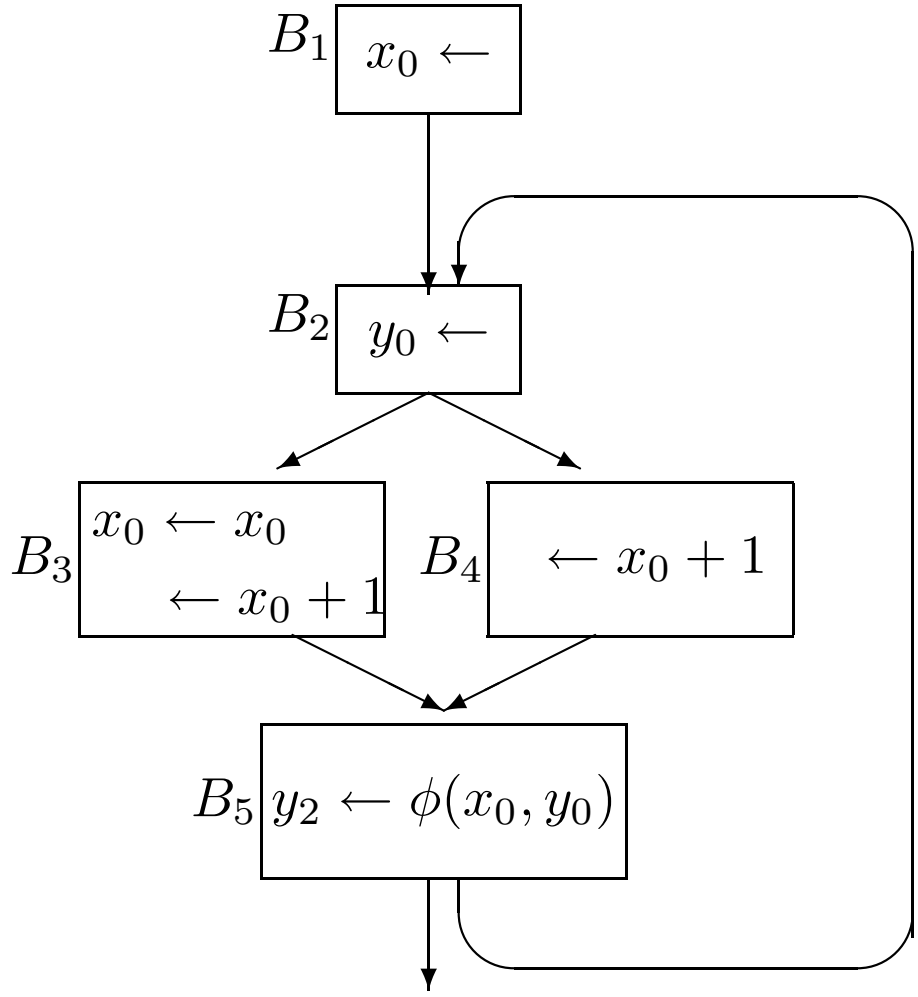
  $\rightarrow$ simpler equations & faster initialization

Retain good code placement

Example

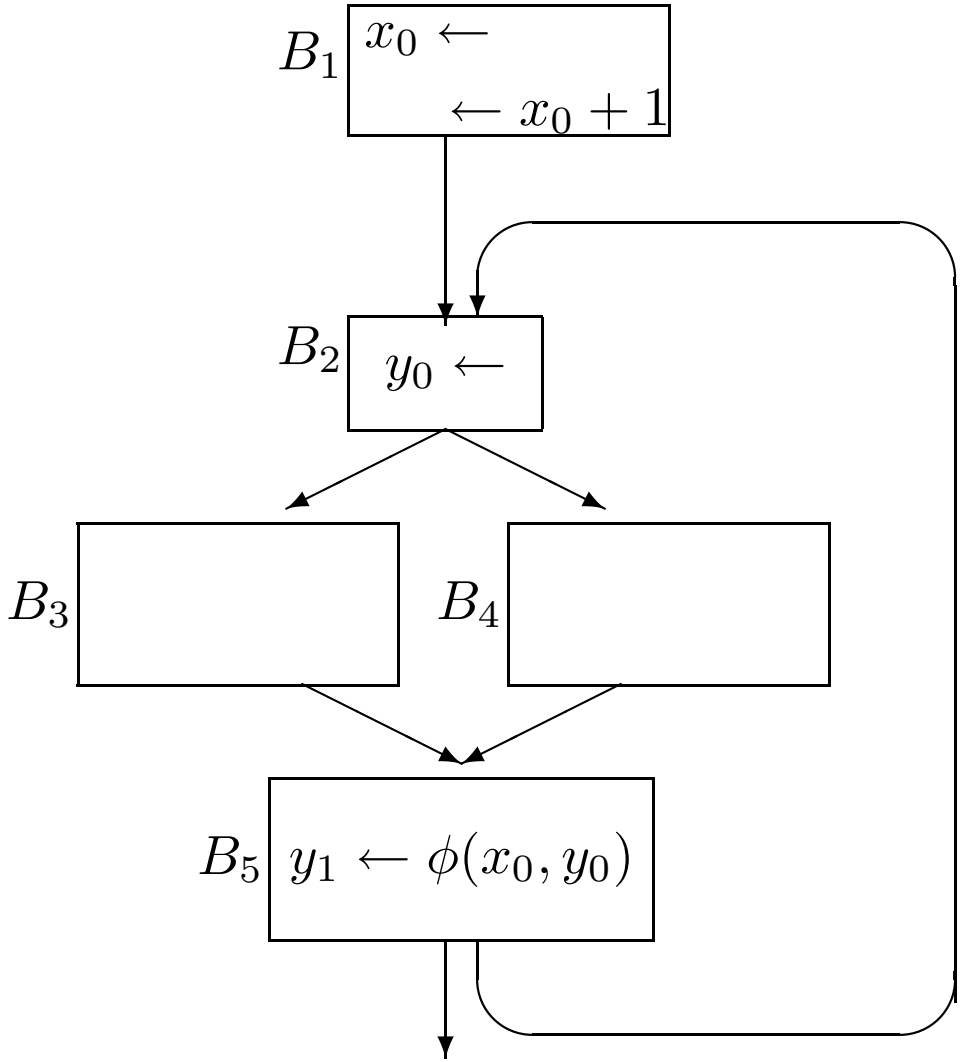


Original



After value numbering

Example (cont'd)



After value-driven code motion

Results: Compile times

Compile-times for value numbering

routine	*blocks*	*SSA names*	*operations*	*DVNT*	*SCC*	*AW&Z*
`tomcatv`	131	3366	3326	0.05	0.08	0.07
`ddeflu`	109	8994	6873	0.11	0.46	0.81
`debflu`	116	7147	4514	0.08	0.48	0.93
`deseco`	251	16502	13121	0.30	0.81	1.85
`twldrv`	261	26913	15168	0.40	3.11	6.09
`fpppp`	2	26588	25934	0.63	0.65	1.16

Compile-times for replacement techniques

	LCM			VDCM		
routine	*set size*	*altered*	*total*	*set size*	*altered*	*total*
`tomcatv`	790	0.39	0.50	924	0.06	0.19
`ddeflu`	1436	1.33	1.47	3648	0.11	0.49
`debflu`	1101	0.76	0.88	3879	0.11	0.49
`deseco`	2513	4.20	4.76	6067	0.51	1.85
`twldrv`	3928	9.26	10.17	16140	0.93	4.77

	GVAL/LCM	SCC/VDCM	NEW *vs* GVAL (%)		GVAL/LCM	SCC/VDCM	NEW *vs* GVAL (%)
gamgen	128618	77018	40.1	bilan	2821	2620	7.1
sgemv	1197	790	34.0	svd	4296	4042	5.9
hmoy	34	23	32.4	solve	192	181	5.7
rkf45	48	35	27.1	debflu	4570	4331	5.2
inter	13	10	23.1	spline	728	690	5.2
inideb	888	747	15.9	sortie	82	78	4.9
paroi	3664	3120	14.8	seval	83	79	4.8
debico	2947	2512	14.8	pastem	3348	3208	4.2
cardeb	884	785	11.2	dyeh	26	25	3.8
drepvi	299	266	11.0	deseco	12155	11724	3.5
drigl	129	115	10.9	ddeflu	722	697	3.5
dcoera	160	144	10.0	fpppp	4056	3922	3.3
colbur	129	117	9.3	si	157	152	3.2
coeray	105	96	8.6	prophy	3587	3479	3.0
lissag	83	76	8.4	inithx	2339	2279	2.6
yeh	134	124	7.5	fmtgen	172	168	2.3
repvid	2596	2406	7.3	heat	180	177	1.7
twldrv	66423556	61588127	7.3	tomcatv	178976655	176884174	1.2

	GVAL/LCM	SCC/VDCM	NEW *vs* GVAL (%)
saxpy	472	467	1.1
efill	214	212	0.9
urand	225	223	0.9
sgemm	838	834	0.5
decomp	639	637	0.3
iniset	38088	38088	0.0
saturr	243	243	0.0
subb	539	539	0.0
supp	693	693	0.0
ihbtr	432	432	0.0
inisla	6	6	0.0
x21y21	219	219	0.0
sigma	48	48	0.0
vgjyeh	20	20	0.0
bilsla	6	6	0.0
arret	19	19	0.0
intowp	5	5	0.0
ilsw	3	3	0.0

	GVAL/LCM	SCC/VDCM	NEW *vs* GVAL (%)
lclear	90	90	0.0
aclear	58	58	0.0
rkfs	279	279	0.0
fehl	515	515	0.0
zeroin	761	761	0.0
integr	2020	2021	0.0
fmin	876	877	-0.1
orgpar	103	104	-1.0
fmtset	338	345	-2.1

	DVNT/LCM	SCC/VDCM	NEW *vs* DVNT (%)		DVNT/LCM	SCC/VDCM	NEW *vs* DVNT (%)
coeray	100	96	4.0	x21y21	219	219	0.0
paroi	3226	3120	3.3	colbur	117	117	0.0
svd	4113	4042	1.7	si	152	152	0.0
repvid	2432	2406	1.1	heat	177	177	0.0
pastem	3234	3208	0.8	debflu	4331	4331	0.0
twldrv	61713295	61588127	0.2	hmoy	23	23	0.0
ddeflu	698	697	0.1	debico	2512	2512	0.0
tomcatv	176889274	176884174	0.0	inideb	747	747	0.0
iniset	38088	38088	0.0	sigma	48	48	0.0
saturr	243	243	0.0	vgjyeh	20	20	0.0
subb	539	539	0.0	yeh	124	124	0.0
supp	693	693	0.0	drigl	115	115	0.0
inter	10	10	0.0	cardeb	785	785	0.0
prophy	3479	3479	0.0	bilan	2620	2620	0.0
lissag	76	76	0.0	drepvi	266	266	0.0
ihbtr	432	432	0.0	dyeh	25	25	0.0
inisla	6	6	0.0	dcoera	144	144	0.0
inithx	2279	2279	0.0	deseco	11724	11724	0.0

	DVNT/LCM	SCC/VDCM	NEW *vs* DVNT (%)
sortie	78	78	0.0
integr	2021	2021	0.0
bilsla	6	6	0.0
orgpar	104	104	0.0
arret	19	19	0.0
intowp	5	5	0.0
ilsw	3	3	0.0
lclear	90	90	0.0
aclear	58	58	0.0
fmtset	345	345	0.0
fmtgen	168	168	0.0
gamgen	77018	77018	0.0
efill	212	212	0.0
fpppp	3922	3922	0.0
saxpy	467	467	0.0
sgemv	790	790	0.0
sgemm	834	834	0.0
rkfs	279	279	0.0

	DVNT/LCM	SCC/VDCM	NEW *vs* DVNT (%)
rkfs	279	279	0.0
rkf45	35	35	0.0
fehl	515	515	0.0
spline	690	690	0.0
seval	79	79	0.0
solve	181	181	0.0
decomp	637	637	0.0
urand	223	223	0.0
zeroin	761	761	0.0
fmin	873	877	-0.5

Results: Summary

Compared three techniques

- DVNT/LCM, GVAL/LCM, and SCC/LCM
- used 63 Fortran routines from various benchmarks
- metric is number of instructions executed
- ignores register pressure

Quick summary

1. SCC/VDCM beat GVAL/LCM 41 times (max 40%)
2. GVAL/LCM beat SCC/VDCM 3 times (max 2%)
3. SCC/VDCM beat DVNT/LCM 7 times (max 4%)
4. DVNT/LCM beat SCC/VDCM 1 time (max .5%)

$\Rightarrow$ should use one of the hash-based techniques !