Debugging of Globally Optimized Programs Using Data Flow Analysis*

Roland Wismüller
Lehrstuhl für Rechnertechnik und Rechnerorganisation
Technische Universität München (Munich Univ. of Technology)
Fakultät für Informatik (Dept. of Computer Science)

24.06.1994

* Partly funded by the German Science Foundation, Contract SFB342, TP A1

Why Debug Optimized Programs?

Automatic code transformation (optimization) is necessary for

- high performance on RISCs
- parallel architectures (superscalar, VLIW, multiprocessors)

$\Rightarrow$ production code should be optimized code

But:

- production code has bugs
- debugging unoptimized versions is not always adequate

$\leadsto$ need to recompile

$\leadsto$ no post-mortem debugging

$\leadsto$ need to reproduce the bug

sporadic bugs, unmasked bugs, time constraints, optimization errors

```
...
for (i=0;i<10;i++) {
   if (a[i] > 0) {
      y = y + a[i]
      f = g + h
      w = a[i] * 2
      z = z + w
      x = a[i] + 1
   }
   else {
      w = a[i] * 2
      x = -a[i] + 1
      z = z - w
      y = y + a[i]
      f = g + h
   }
}
```

```
      ...
      mov   0,%i3          ! i = 0
      add   %o0,%l5,%l6    ! f = g + h
      add   %fp,-48,%i2    ! i2 = &a[i]
                           ! loop:
L1:   ld    [%i2],%i5      !    i5 = a[i]
      tst   %i5
      ble   L2
      sll   %i5,1,%i1      !    w = a[i] * 2
                           !    if (a[i] > 0)
      b     L3
      add   %i4,1,%i4      !       z = z + w
                           !    else
L2:   sub   %i4,1,%i4      !       z = z - w
L3:   inc   %i3            !    i++
      cmp   %i3,10
      add   %l7,%i5,%l7    !    y = y + a[i]
      bl    L1
      inc   4,%i2          !    i2 = &a[i]
                           !    if (i<10) goto loop
```

Analysis to Determine Values of Variables

Objectives:

- find register assigned to a variable
- determine whether the register contains the variable's expected value

Requirements:

- independent of optimization details (analyze sum transformation)
- usable with any breakpoint mapping
- ability to use run time information
- usable for real programs (handle pointer assignments)

Restrictions:

- no loop transformations

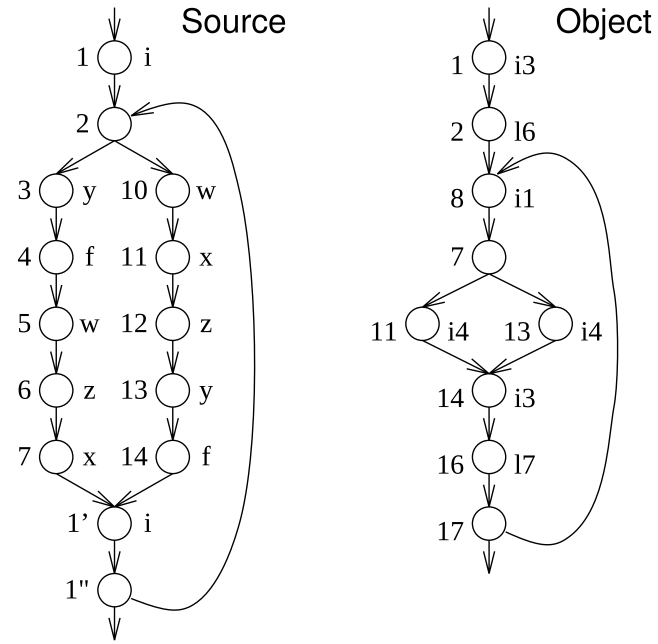
Compiler-Debugger-Interface

Flow graphs for source and object code

- operations that modify program state
- modified variables

Mapping between source and object code

- equivalent assignments
  $\rightsquigarrow$ write same values
- corresponding edges
  $\rightsquigarrow$ equivalent control flow operations



Source	Object
1	1
1’	14
3,13	16
4,14	2
5,10	8
6	11
12	13

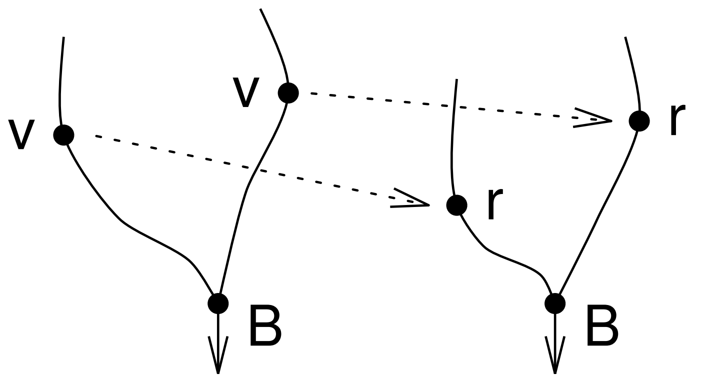
Source	Object
<2,3>	<7,11>
<2,10>	<7,13>
<7,1’>	<11,14>
<14,1’>	<13,14>
<1,2>	<2,8>
<1",2>	<17,8>

Finding a Variable's Assigned Register

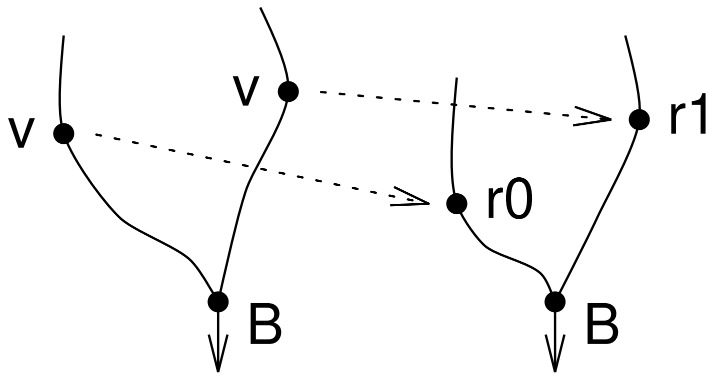
Consider all reaching definitions of v:

Source Object

- all corresponding stores write to the same register r:
  - r is the register assigned to v



- some definition has no corresponding store, or
- some corresponding store writes to a different register:
  - no (unique) associated register
  - variable v is not current



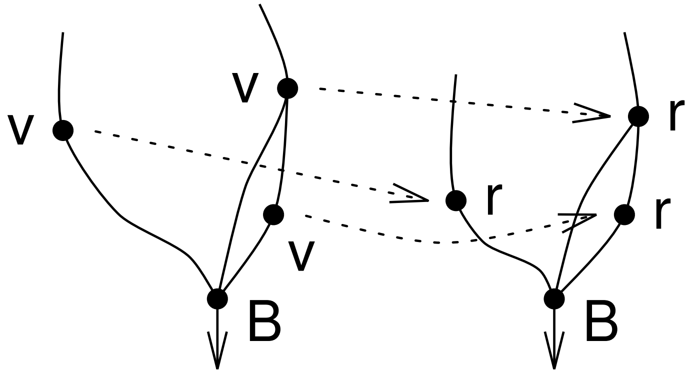
Analyzing Currentness

Consider all reaching definitions of v:

Source Object

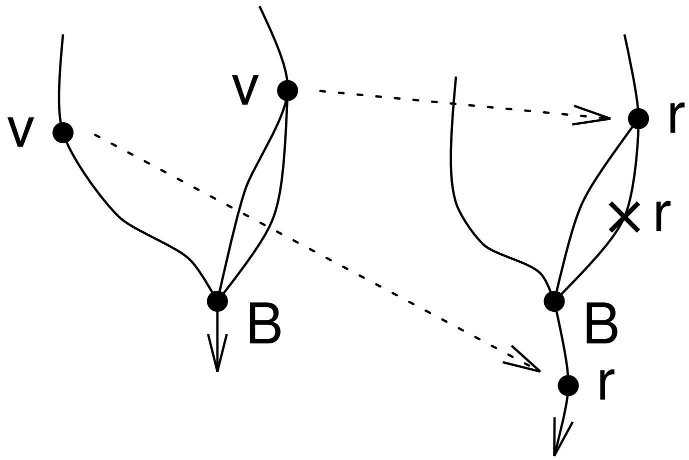
v is current in r, if:

- all corresponding stores to r reach the object breakpoint along all proper paths



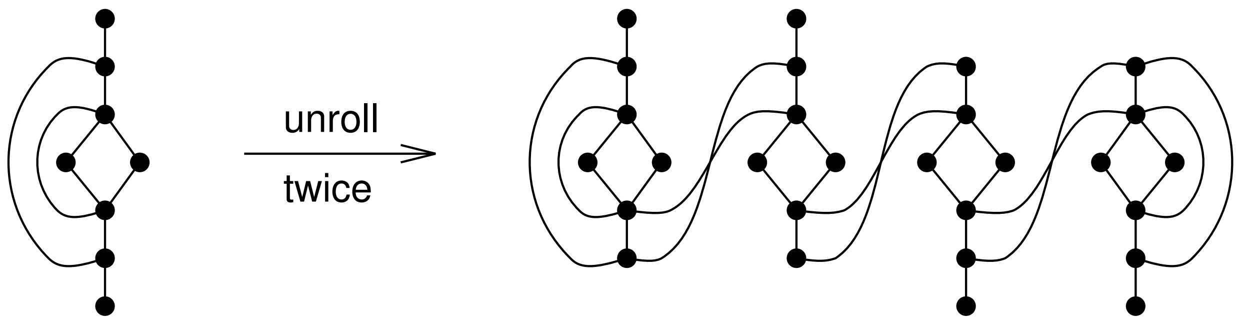
if v is not current in r, then:

- some corresponding store doesn't reach the breakpoint along some proper path



Problems to Solve

Distinguish between instances of a definition
Restict paths in object flow graph
$\Longrightarrow$ unrolled graphs



Handle indirect assignments $\Longrightarrow$ recursive analysis

1: a = x	1: a = x
2: *p = y	2: *q = z
3: *q = z	3: *p = y
4: *r = 0	4: *r = 0
5: . . .	5: . . .

1. analyze definition 4 at (5,5) ✓
2. analyze definition 3 at (4,4) ϟ
   (instr. 3 may kill instr. 2)

Algorithm

1. Determine the variable's register
2. For $m=0,\ldots,3\cdot l+2$:

   For each source definition reaching the breakpoint along m back edges:

   a) unroll flow graphs

   b) restrict paths in object graph according to possible executions

   c) determine object nodes representing possible breakpoint instances

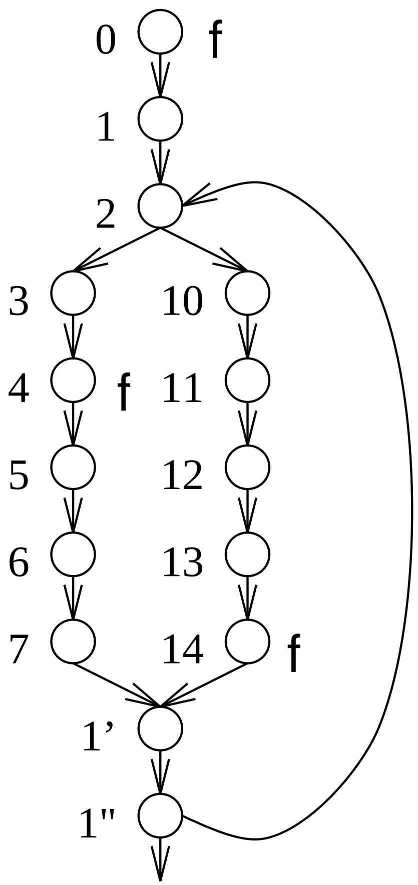
   d) determine object nodes that store the variable's expected value

   e) test whether this value is available at the breakpoint along all paths in the object graph
3. For indirect assignments reaching the breakpoint:

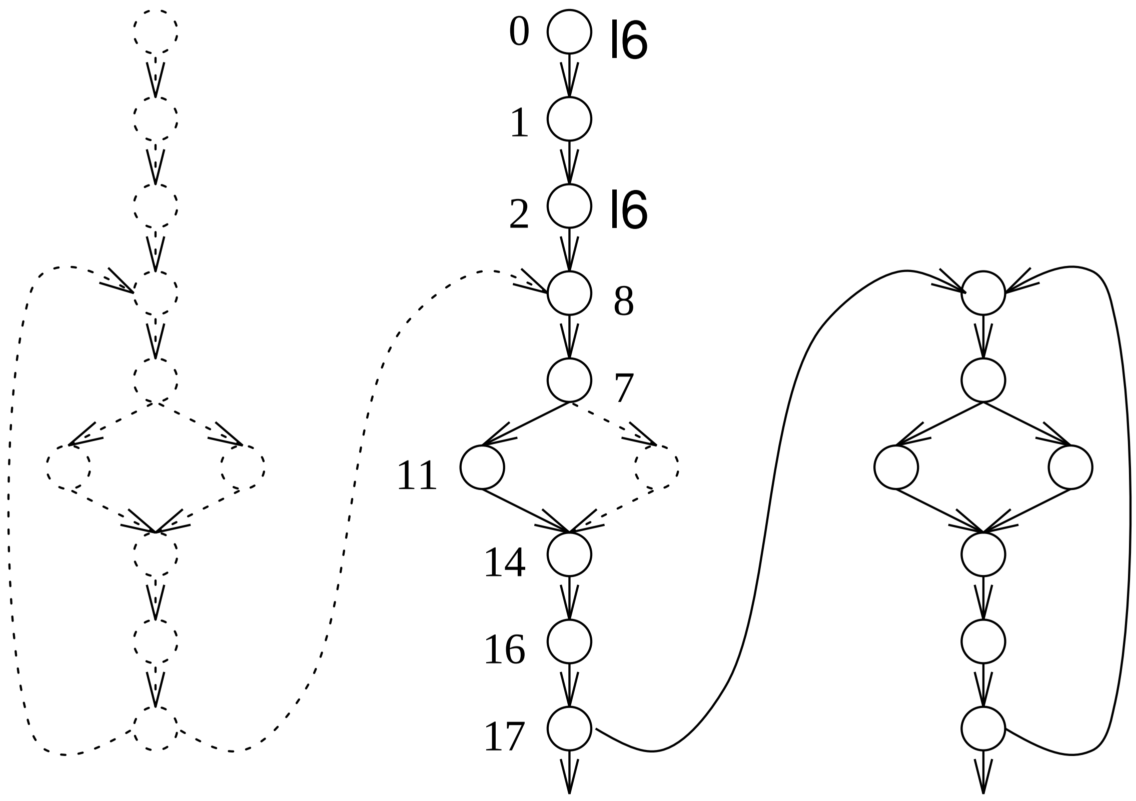
   perform recursive analysis

Example: Testing Definition 0 of Variable f

Source graph:

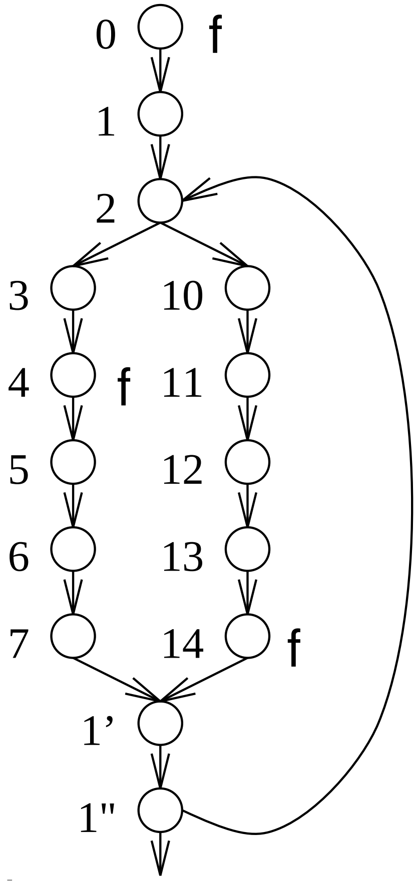


Unrolled object graph:

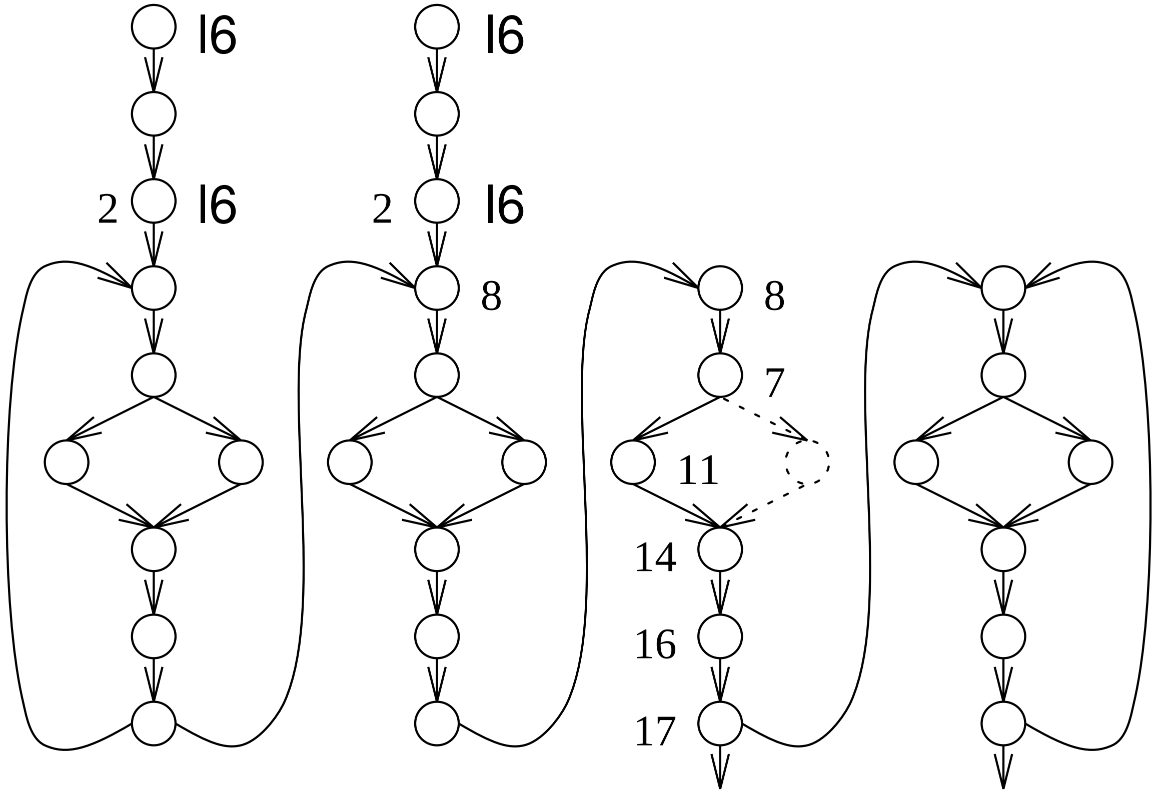


Example: Testing Definition 14 of Variable f

Source graph:



Unrolled object graph:



Implementation

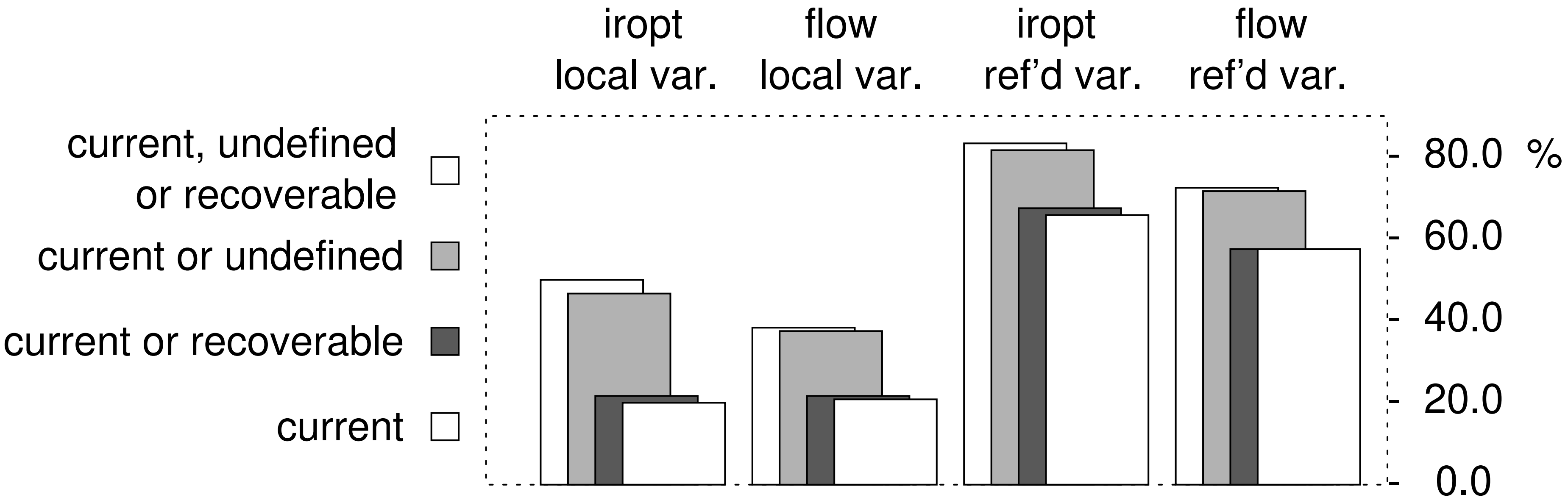
SUN C-Compiler for Sparc (SC1.0)

- implementation costs:
  - $\leadsto$ existing compiler: 94000 lines of code
  - $\leadsto$ additional code: 2200 lines
  - $\leadsto$ mappings can be updated locally in each optimization step
- compilation cost for debugging support:
  - $\leadsto$ no optimization: +19% time, +550% file size
  - $\leadsto$ with optimization: +29% time, +2000% file size

Analysis algorithm

- module to gather data on frequency of noncurrent variables
- integrated into debugger DETOP

Some Results



Percentage of breakpoints where all variables are in the indicated class

Conclusion

- analysis is possible in a general, compiler idependent way
- compiler support is simple
- results suggest that useful debugger behavior is often possible

Future Work

- get user experience
- algorithms for breakpoint mapping
- loop transformations, parallelization