COMP 512
Rice University
Spring 2015

Operator Strength Reduction

— the Vick-Simpson algorithm —

Citation numbers refer to entries in the EaC2e bibliography.

Operator Strength Reduction

The Algorithmic Plan

- Capitalize on the properties of **SSA** form
- Find **SCC**s in the **SSA** graph
  - ♦ Each non-trivial **SCC** might be an **IV**
    - → *Test the **SCC** as it is discovered, so we need a cheap test*
    - → *Discover **RC**s relative to the **SCC** with a cheap test*
  - ♦ Reduce operations on the fly
    - → *Recognize candidates for reduction with a cheap test*
    - → *Use structural information (e.g., **DOM**) to place new computations*
  - ♦ Accumulate information for linear function test replacement
- Use results of prior transformations
  - ♦ Assume constant propagation and code motion
  - ♦ Use **DOM** information from **SSA** construction

Cheap means **O**(1) if possible

Operator Strength Reduction

Review from last lecture

Consider the following simple loop

```
sum = 0
do i = 1 to 100
    sum = sum + a(i)
end do
```

	loadI	0	$\Rightarrow r_{sum}$
	loadI	1	$\Rightarrow r_i$
	loadI	100	$\Rightarrow r_{100}$
loop:	subI	r_i,1	$\Rightarrow r_1$
	multI	r_1,4	$\Rightarrow r_2$
	addI	r_2,@a	$\Rightarrow r_3$
	load	r_3	$\Rightarrow r_4$
	add	r_4,r_{sum}	$\Rightarrow r_{sum}$
	addI	r_i,1	$\Rightarrow r_i$
	cmp_LT	r_i,r_{100}	$\Rightarrow r_5$
	cbr	r_5	$\rightarrow$ loop, exit
exit: ...			

address of a(i) (subI, multI, addI)

What's wrong with this picture?

- Takes 3 operations to compute the address of a(i)
- On some machines, integer multiply is slow

This lecture works from the same example as the lecture on Cocke-Kennedy, so we will quickly review the example.

Operator Strength Reduction

Review from last lecture

Consider the value sequences taken on by the various registers

	loadI	0	$\Rightarrow r_{sum}$	
	loadI	1	$\Rightarrow r_i$	$r_{sum} = \bot$
	loadI	100	$\Rightarrow r_{100}$	$\mathbf{r_i}$ = { 1, 2, 3, 4, … }
loop:	subI	r_i,1	$\Rightarrow r_1$	r_{100} = { 100 }
	multI	r_1,4	$\Rightarrow r_2$	$\mathbf{r_1}$ = { 0, 1, 2, 3, … }
	addI	r_2,@a	$\Rightarrow r_3$	$\mathbf{r_2}$ = { 0, 4, 8, 12, … }
	load	r_3	$\Rightarrow r_4$	$\mathbf{r_3}$ = { @a, @a+4, @a+8, @a+12, … }
	add	r_4,r_{sum}	$\Rightarrow r_{sum}$	$r_4 = \bot$
	addI	r_i,1	$\Rightarrow r_i$	
	cmp_LT	r_i,r_{100}	$\Rightarrow r_5$	$r_5 = \bot$
	cbr	r_5	$\rightarrow$ loop, exit	
exit: …				

r_i, r_1, r_2, and r_3 take on predictable sequences of values

- r_1 and r_2 are intermediate values, while r_3 and r_i play important roles
- We can compute them cheaply & directly

Operator Strength Reduction

Review from last lecture

Computing r_3 directly yields the following code

	loadI	0	$\Rightarrow r_{sum}$
	loadI	1	$\Rightarrow r_i$
	loadI	100	$\Rightarrow r_{100}$
	loadI	@a	$\Rightarrow r_3$
loop:	load	r_3	$\Rightarrow r_4$
	addI	r_3, 4	$\Rightarrow r_3$
	add	r_4, r_{sum}	$\Rightarrow r_{sum}$
	addI	r_i,1	$\Rightarrow r_i$
	cmp_LT	r_i, r_{100}	$\Rightarrow r_5$
	cbr	r_5	$\rightarrow$ loop,exit
exit: ...			

address of a(i)

r_3 = { @a, @a+4, @a+8, @a+12, ... }

Still, we can do better ...

- From 8 operations in the loop to 6 operations
- No expensive multiply, just cheap adds

Operator Strength Reduction

Review from last lecture

Shifting the loop's exit test from r_i to r_3 yields

loadI 0 ⇒ r_{sum}
loadI @a ⇒ r_3
addI r_3,396 ⇒ r_{lim}
loop: load r_3 ⇒ r_4
addI r_3, 4 ⇒ r_3
add r_4,r_{sum} ⇒ r_{sum}
cmp_LT r_3,r_{lim} ⇒ r_5
cbr r_5 → loop,exit
exit: ...

r_3 = { @a, @a+4, @a+8, @a+12, … }

- Address computation went from -,+,* to +
- Exit test went from +, <u>cmp</u> to <u>cmp</u>
- Loop body went from 8 operations to 5 operations
  - ♦ Got rid of that expensive multiply, too

Pretty good speedup on most machines

37.5% of ops in the loop, even if mult takes one cycle

Not redundant or invariant

Operator Strength Reduction

And, as an aside, unrolling also helps

	loadI	0	$\Rightarrow r_{sum}$	
	loadI	@a	$\Rightarrow r_3$	
	addI	r_3,396	$\Rightarrow r_{lim}$	
loop:	load	r_3	$\Rightarrow r_4$	**Copy # 1 of loop body**
	addI	r_3, 4	$\Rightarrow r_3$	
	add	r_4,r_{sum}	$\Rightarrow r_{sum}$	
	load	r_3	$\Rightarrow r_4$	**Copy # 2 of loop body**
	addI	r_3, 4	$\Rightarrow r_3$	
	add	r_4,r_{sum}	$\Rightarrow r_{sum}$	
	cmp_LT	r_3,r_{lim}	$\Rightarrow r_5$	**Shared test & branch**
	cbr	r_5	$\rightarrow$ loop,exit	
exit: ...				

Now, 8 operations for 2 iterations, or 50% of the operations and a smaller percentage of the cycles (due to elimination of multiplies)

Operator Strength Reduction

New material !

Assumptions for the OSR Algorithm

- Low-level **IR**, such as **ILOC**, converted into **SSA** form
- Constant propagation and loop-invariant code motion have been applied

Also important for CK & ACK

Terminology

- A *strongly connected component* (**SCC**) of a directed graph is a region where a path exists from each node to every other node
- A *region constant* (**RC**) of an **SCC** is an **SCC**-invariant value
- An *induction variable* (**IV**) of an **SCC** is one whose value only changes in the **SCC** when operations increment it by an **RC** or an **IV**, or when it is the destination of a **COPY** from another IV
- A *candidate* for reduction is an operation "$x \leftarrow y * z$" where $y, z \in$ **IV** $\cup$ **RC** and either $y \in$ **IV** or $z \in$ **IV**

Intuitively, we are interested in induction variables that are updated in a cyclic fashion. The self-dependence creates the pattern of repetition from which the *strong form* of strength reduction derives its benefits.

The classic papers, e.g., Cocke-Kennedy, and Allen-Cocke-Kennedy, define **IV**s this way. The **OSR** algorithm only finds **IV**s that form a cycle in the **SSA** graph. The practical results are equivalent.

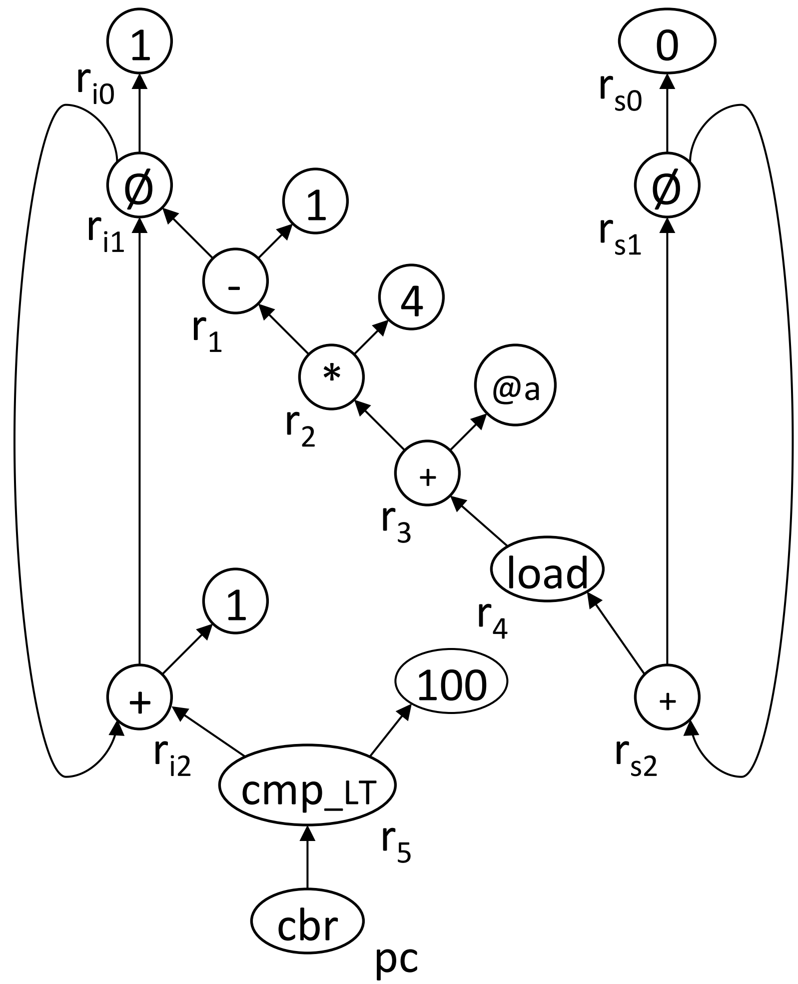
Operator Strength Reduction

Our example in semi-pruned SSA Form

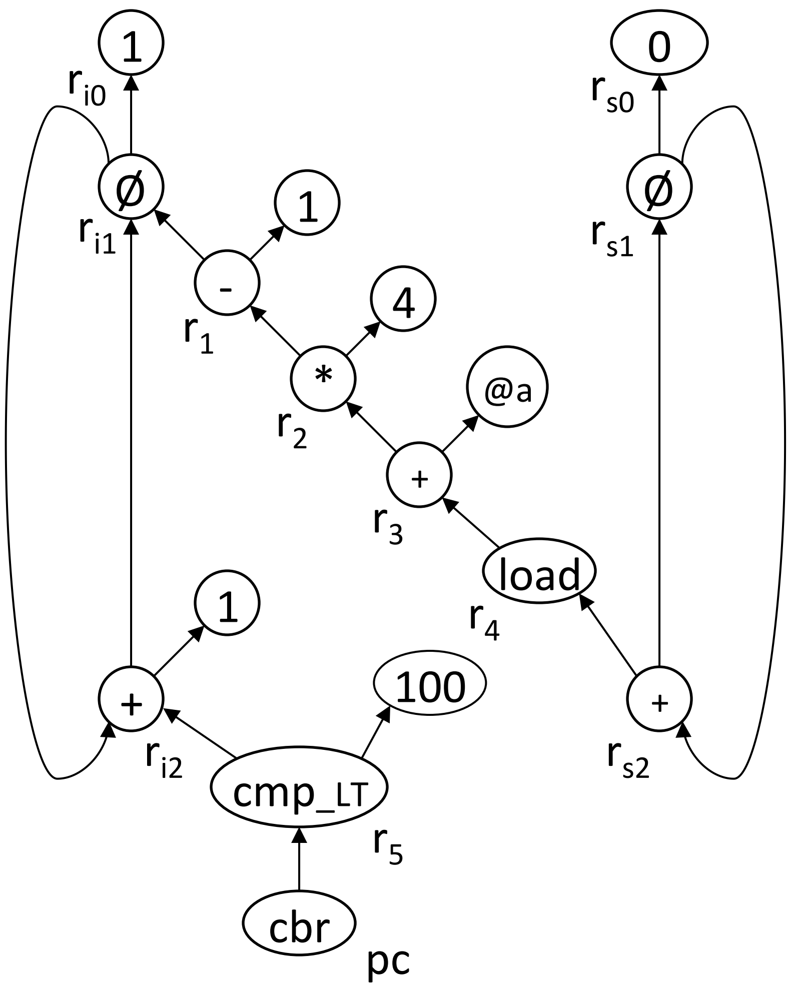
	loadI	0	⇒ r_{s0}
	loadI	1	⇒ r_{i0}
	loadI	100	⇒ r_{100}
loop:	phi	r_{s0},r_{s2}	⇒ r_{s1}
	phi	r_{i0},r_{i2}	⇒ r_{i1}
	subI	r_{i1},1	⇒ r_1
	multI	r_1,4	⇒ r_2
	addI	r_2,@a	⇒ r_3
	load	r_3	⇒ r_4
	add	r_4,r_{s1}	⇒ r_{s2}
	addI	r_{i1},1	⇒ r_{i2}
	cmp_LT	r_{i2},r_{100}	⇒ r_5
	cbr	r_5	→ loop,exit
exit: ...			

Short-lived temporary values

SSA Form as a Graph



Operator Strength Reduction



SSA form as a graph

- Each **IV** is an **SCC**
- Not every **SCC** is an **IV**
- x ∈ **RC** if x is a constant or its definition is in a block that *dominates* the entry of the **SCC**
- Compute **DOM** & **RPO** numbers for the **SSA** graph

Using SSA as a graph simplifies OSR

- Find **IV**s with **SCC** finder
- Test operations in **SCC**
- Constant time test for **RC**
  - > Constant or test with **DOM**

Prior algorithms used multiple passes over the **IR**, inner loop to outer loop..

Operator Strength Reduction

Finding SCCs

- Use Tarjan's algorithm
- Well-understood method
- Takes **O**(*N+E*) time

Useful property

- **SCC** popped only after all its external operands have been popped
- Reduce the **SCC**s as popped
  - ♦ |**SCC**| > 1 ⇒ if its an **IV**, mark it
  - ♦ |**SCC**| = 1 ⇒ try to reduce it
- We only need to add one line

```
DFS(n)
   n.DFSnum ← nextDFSnum++
   n.visited ← true
   n.low ← n.DFSnum
   push(n)
   for each o ∈ { operands of n}
      if o.visited = false then
         DFS(o)
         n.low ← min(n.low, o.low)
      if o.DFSnum < n.DFSnum and
         o ∈ stack then
         n.low ← min(n.low, o.DFSnum)
   if n.low = n.DFSnum then
      SCC ← { }
      until x = n do
         x ← pop()
         SCC ← SCC ∪ { x }
         Process(SCC)
```

Operator Strength Reduction

What should Process(*r*) do?

- If *r* is one node, try to reduce it
- If *r* is a collection of nodes
  - Check to see if it is an **IV**
  - If so, reduce it & any ops that use it
  - If not, try to reduce the ops in *r*

```
Process(r)
  if r has only one member, n then
    if n has the form x ← IV x RC, x ← RC x IV,
      x ← IV ± RC, or x ← RC + IV then
      Replace(n,IV,RC)
    else n.header ← NULL
  else ClassifyIV(r)
```

Let's tackle the easier problem first – ClassifyIV()

Operator Strength Reduction

```
ClassifyIV(r)
   header ← first(r)
   for each node n ∈ r
      if header→RPOnum > n.block →RPOnum then
         header ← n.block
   for each node n ∈ r
      if n.op is not one of { Ø, +, -, COPY } then
         r is not an induction variable
      else
         for each o ∈ { operands of n }
            if o ∉ r and not RCon(o,header) then
               r is not an induction variable
   if r is an induction variable then
      for each node n ∈ r
         n.header ← header
   else
      for each node n ∈ r
         if n has the form x ← IV x RC, x ← RC x IV, x ← IV ± RC, or x ← RC + IV then
            Replace(n,IV,RC)
         else  n.header ← NULL
```

Find SCC header by CFG RPO #

Eliminate SCCs as IVs

Mark SCC as an IV

Reduce these ops

```
Rcon(o,header)
   if o.op is loadI  /* constant */
      then return true
      else if o.block >> header
         then return true
         else return false
```

>> means "strictly dominates"

Operator Strength Reduction

search and **add** deal with the hash table

```
/* replace n with a COPY */
Replace(n,iv,rc)
    result ← Reduce(n.op,iv,rc)
    Replace n with COPY from result
    n.header ← iv.header

/*  create new IV & return its name */
Reduce(op,iv,rc)
    result ← search(op,iv,rc)
    if result is not found then
        result ← a new name
        add(op,iv,rc,result)
        newDef ← copyDef(iv,result)

        for each operand o of newDef
            if o.header = iv.header then
                replace o with Reduce(op,o,rc)
            else if (opcode = x or newDef.op = Ø) then
                replace o with Apply(op,o,rc)

    return result
```

Replace rewrites op *n* with a COPY operation from its reduced counterpart. It calls **Reduce** to create that counterpart, if necessary.

Returns name of op applied to iv and rc

Clones the definition

Args defined outside SCC ⇒ initial value or the increment

Operator Strength Reduction

```
/* replace n with a COPY */
Replace(n,iv,rc)
    result ← Reduce(n.op,iv,rc)
    Replace n with COPY from result
    n.header ← iv.header

/*  create new IV & return its name */
Reduce(op,iv,rc)
    result ← search(op,iv,rc)
    if result is not found then
        result ← a new name
        add(op,iv,rc,result)
        newDef ← copyDef(iv,result)

        for each operand o of newDef
            if o.header = iv.header then
                replace o with Reduce(op,o,rc)
            else if (opcode = x or newDef.op = Ø) then
                replace o with Apply(op,o,rc)

    return result
```

The Big Picture

- Reduce() creates a new IV, with appropriate range & increment
- In the example, r_3 would range from @a to @a+396, with an increment of 4
- Replace takes a candidate operation and rewrites it with a COPY from the new IV. It uses Reduce to create the IV.

Net effect: replace (i-1)*4+@a with a COPY from some new IV that runs from @a to @a+396 & increments by 4 on each iteration

Operator Strength Reduction

```
/* insert a new operation */
Apply(op,arg1,arg2)
   result ← search(op,arg1,arg2)
   if result is not found then
      if (arg1.header ≠ NULL   /* ∈ IV */
         & RCon(arg2,arg1.header) then
         result ← Reduce(op,arg1,arg2)
      else if arg2.header ≠ NULL   /* ∈ IV */
         & RCon(arg1,arg2.header) then
         result ← Reduce(op,arg2,arg1)
      else
         result ← a new name
         add(op,arg1,arg2,result)
         Choose a location to insert op
         Try constant folding
         Create newOp at the location
         newOp.header ← NULL
   return result
```

The Big Picture

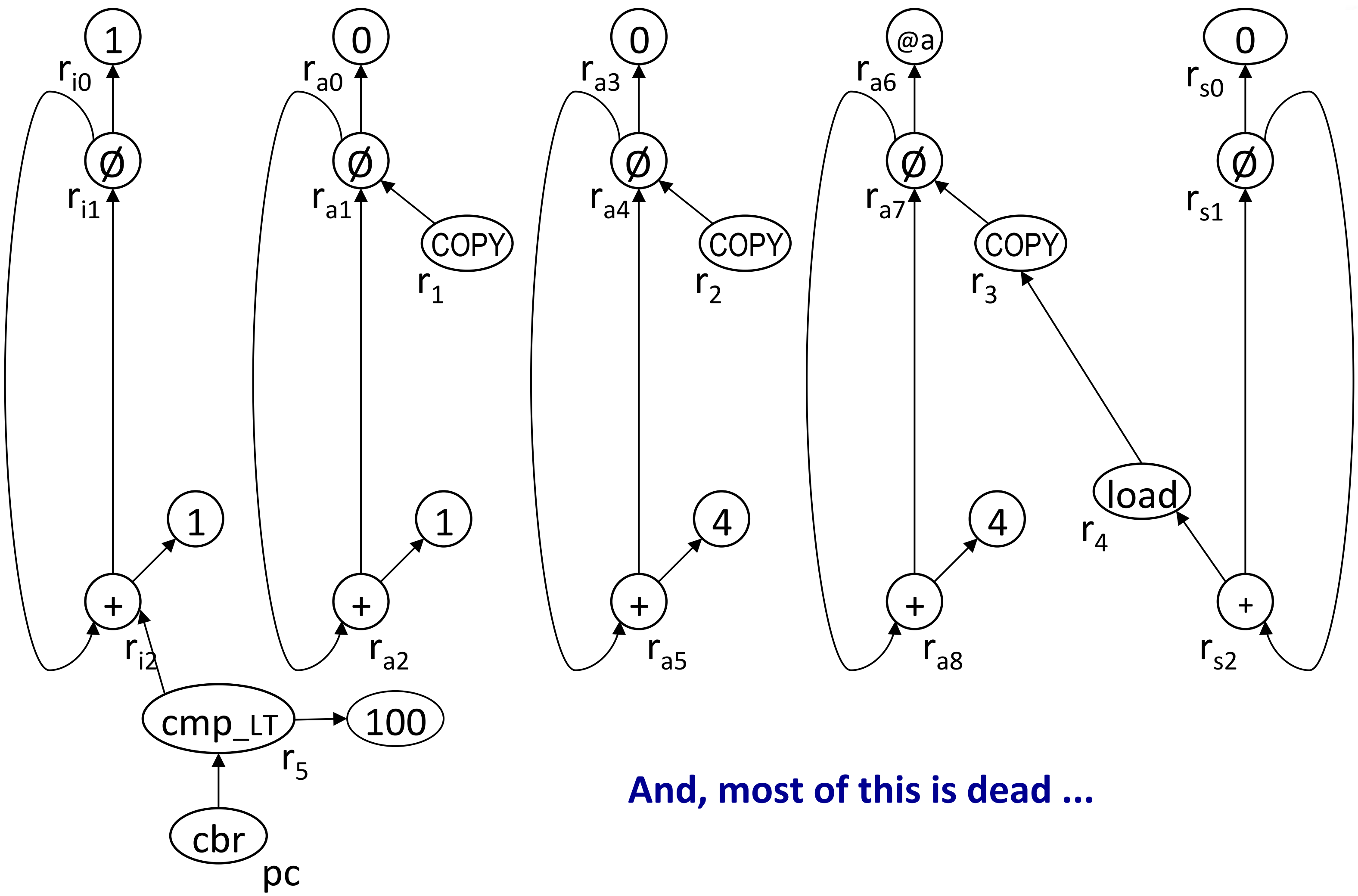
- Apply takes an op & 2 args and inserts the corresponding operation into the code

  (if it isn't already there).
- Uses >> on arg1 & arg2 to find a location

  — does not use landing pad

  — may insert farther away
- Tries to reduce the operation
- Tries to simplify the operation

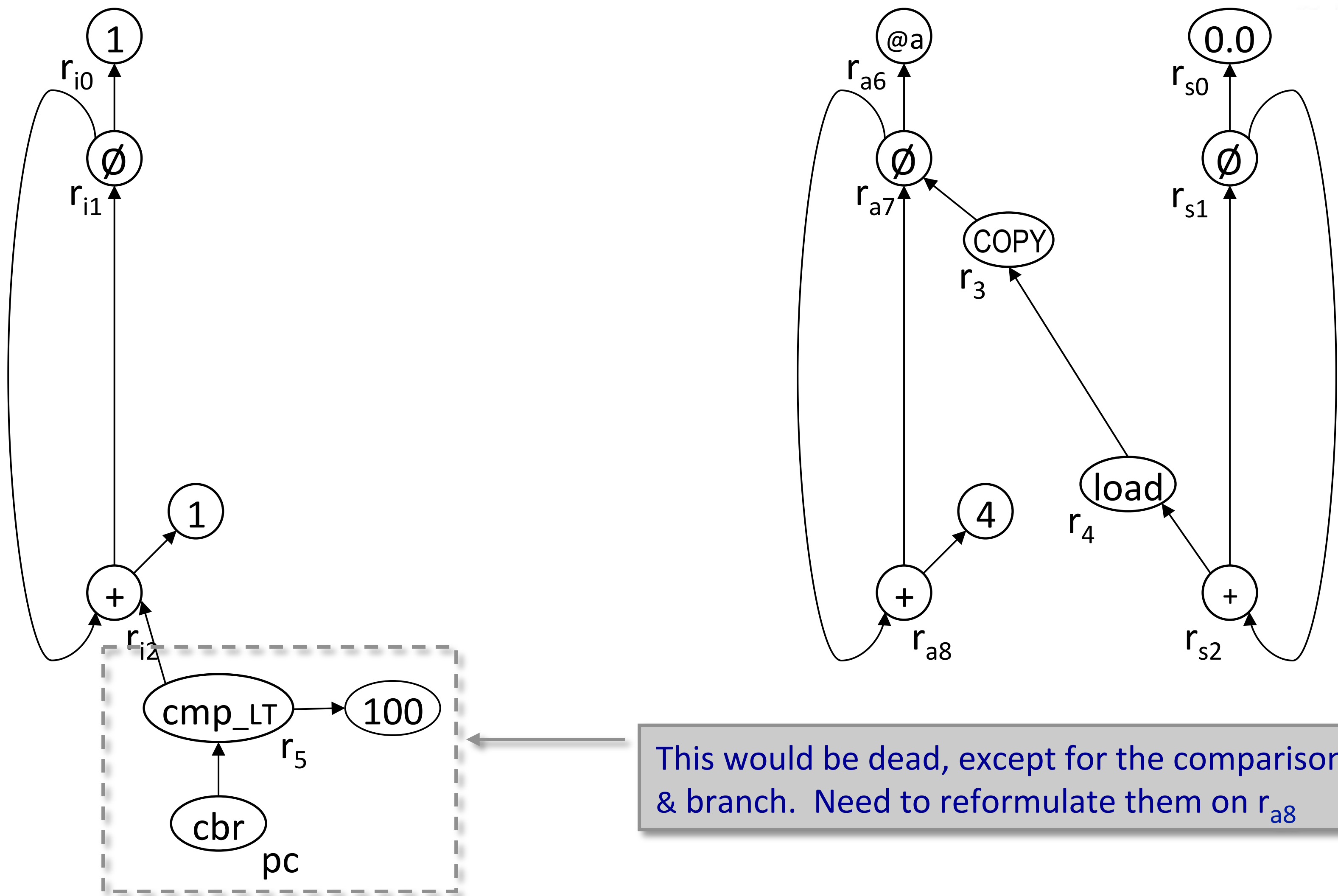
Example



And, most of this is dead ...

Example

The transformation to perform this simplification is called *linear function test replacement.*



This would be dead, except for the comparison & branch. Need to reformulate them on r_{a8}

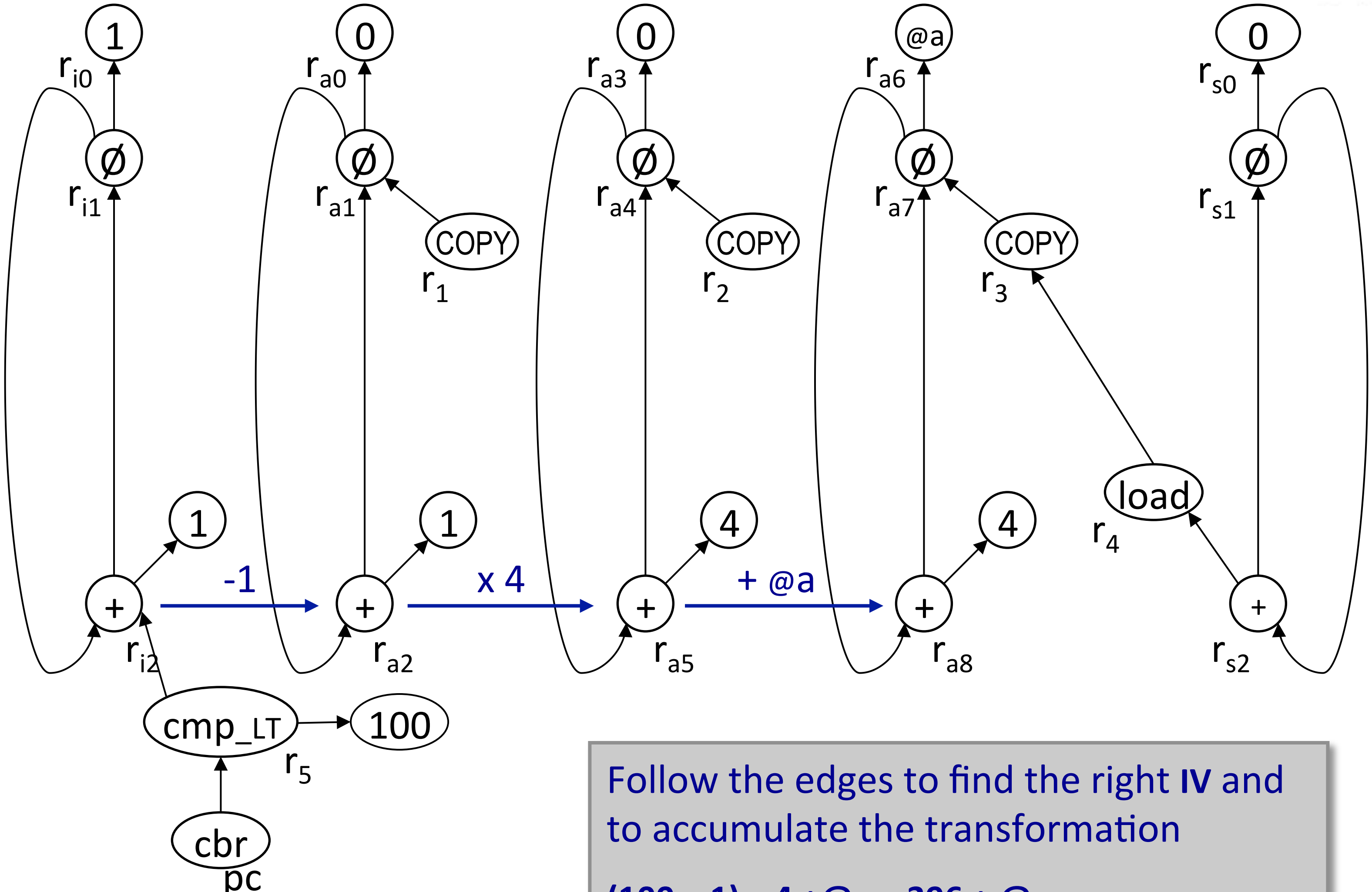
*

Linear Function Test Replacement

Each time a new, reduced IV is created

- Add an **LFTR** edge from old **IV** to new **IV**
- Label edge with the opcode and **RC** of the reduction
- Walk the **LFTR** edges to accumulate the transformation
- Use transformation to rewrite the test

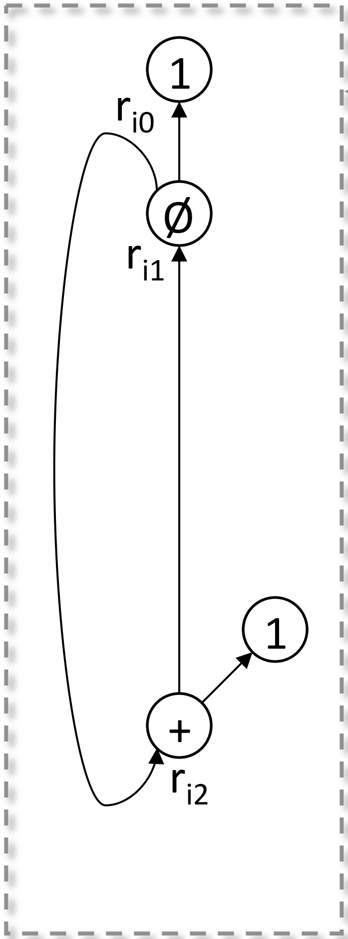
Example



Follow the edges to find the right **IV** and to accumulate the transformation

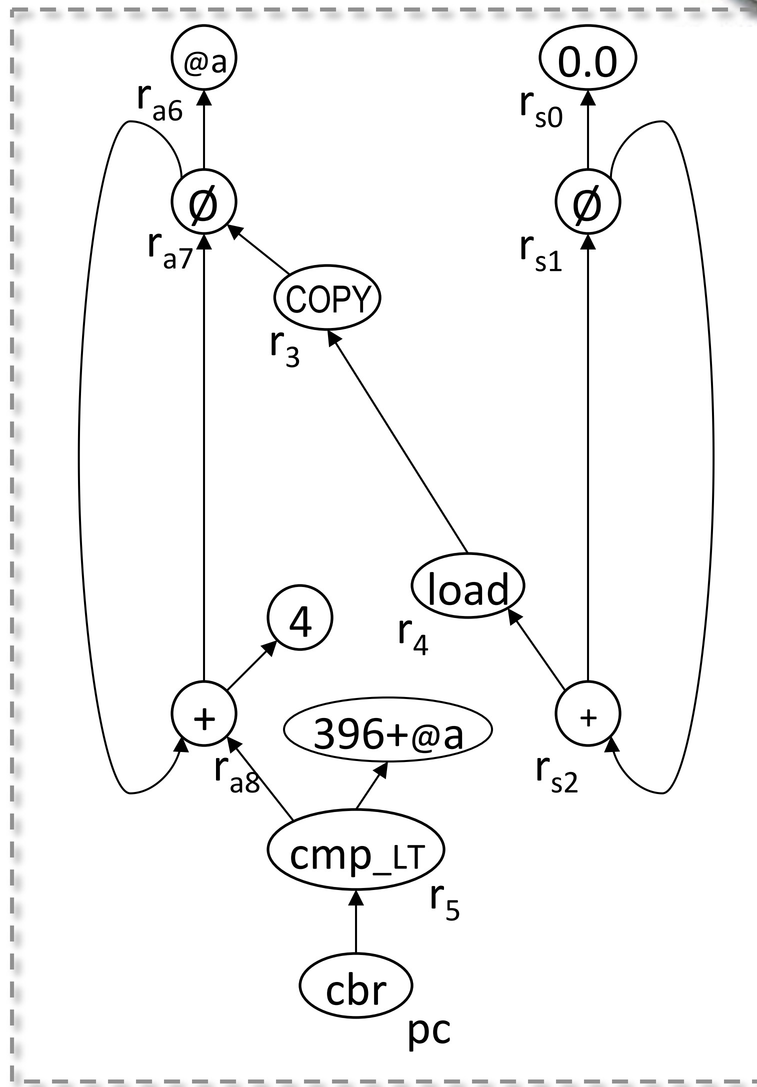
(100 – 1) x 4 +@a = 396 + @a

Example

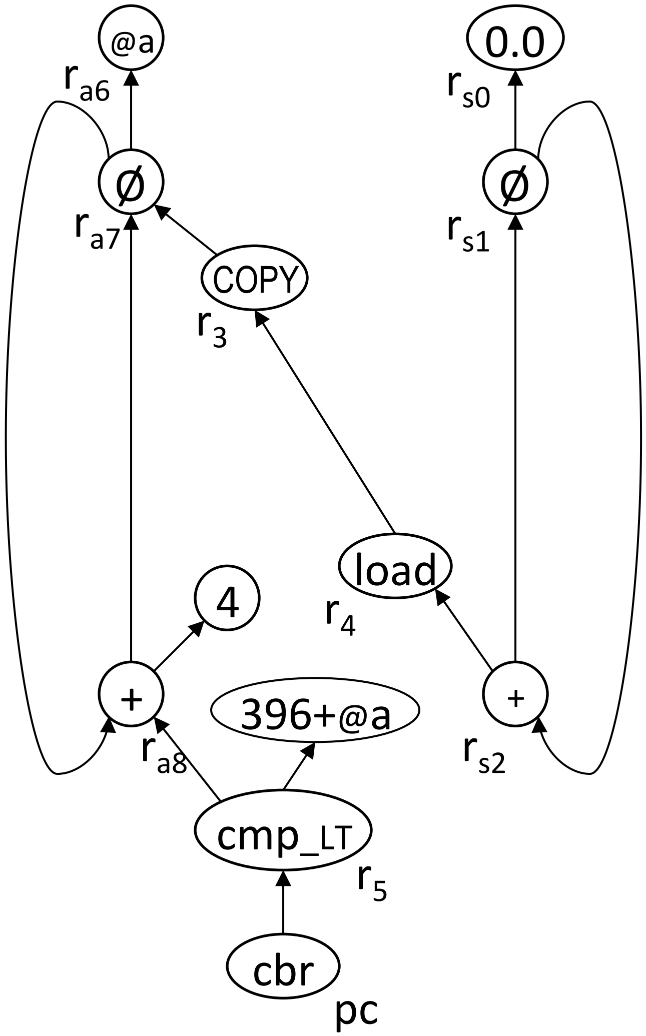


Now, this code is dead.

Not dead!



Example



	loadI	0	$\Rightarrow r_{sum}$
	loadI	@a	$\Rightarrow r_3$
	addI	r3,396	$\Rightarrow r_{lim}$
loop:	load	r_3	$\Rightarrow r_4$
	addI	r_3, 4	$\Rightarrow r_3$
	add	r_4, r_{sum}	$\Rightarrow r_{sum}$
	cmp_LT	r_3, r_{lim}	$\Rightarrow r_5$
	cbr	r_5	$\rightarrow$ loop, exit
exit: ...			

And, we're done ..

Complexity

What does OSR + LFTR cost?

- **LFTR** takes time proportional to the length of the **LFTR** edge chain that it follows
- What about **OSR**?
  - ♦ Each cycle it creates clones every node in the cycle
  - ♦ How bad can that get?

Worst-case Example

$i \leftarrow 0$
while(P_0)
  if (P_1) then
    $i \leftarrow i + 1$
    $k \leftarrow i \times c_1$
  if (P_2) then
    $i \leftarrow i + 2$
    $k \leftarrow i \times c_2$
  …
  if (P_n) then
    $i \leftarrow i + n$
    $k \leftarrow i \times c_n$
  end

$i \leftarrow 0;\ t_1 \leftarrow 0;\ t_2 \leftarrow 0; \ldots ;\ t_n \leftarrow 0$
while(P_0)
  if (P_1) then
    $t_1 \leftarrow t_1 + c_1; t_2 \leftarrow t_2 + c_2; \ldots ; t_n \leftarrow t_n + c_n$
    $i \leftarrow i + 1$
    $k \leftarrow t_{1;}$
  if (P_2) then
    $t_1 \leftarrow t_1 + 2\mathrm{x}\ c_1; t_2 \leftarrow t_2 + 2 \times c_2; \ldots ;$
    $t_n \leftarrow t_n + 2 \times c_n; i \leftarrow i + 2$
    $k \leftarrow t_2$
  …
  if (P_n) then
    $t_1 \leftarrow t_1 + n \times c_1; t_2 \leftarrow t_2 + n \times c_2; \ldots ;$
    $t_n \leftarrow t_n + n \times c_n ; i \leftarrow i + n$
    $k \leftarrow t_n$
  end

This code requires a quadratic number of updates

Complexity

What does OSR + LFTR cost?

- **LFTR** takes time proportional to the length of the **LFTR** edge chain that it follows
- What about **OSR**?
  - ♦ Each cycle it creates clones every node in the cycle
  - ♦ How bad can that get?
  - ♦ In the worst case, **OSR** must insert a number of updates that is quadratic in the size of the original code
  - ♦ Any strength reduction algorithm must insert the same set of updates, if it is to reduce the computation
    - → If it doesn't, it misses the opportunity
  - ♦ Complexity is **part of the problem**, not part of the solution
- **OSR** is as fast (asymptotically) as others
  - ♦ Constant factor faster than Cocke-Kennedy or Allen-Cocke-Kennedy