Identifying
Wasteful Memory Operations
with
DrCCTProf Clients

Milind Chabbi

“Premature optimization is the root of all evil”

True or False?

Stop Misquoting Donald Knuth!

"Premature optimization is the root of all evil”

- Commonly misinterpreted as:
  - Constant factors don’t matter
  - Micro-optimization is a waste of time
  - Engineering time costs more than CPU time
  - The machine’s so fast it won’t matter
  - Nobody will notice an occasional delay
  - We can just buy more servers

Stop Misquoting Donald Knuth!

"Premature optimization is the root of all evil"

- Commonly misinterpreted as:
  - Constant factors don't matter
  - Micro-optimization is a waste of time
  - Engineering time costs more than CPU time
  - The machine's so fast it won't matter
  - Nobody will notice an occasional delay
  - We can just buy more servers

Hope to debunk this myth

Performance Complacency

- Waste creeps in little by little
  - An extra string copy, an extra string to int conversion
  - Bound checks
  - Too many allocs
  - Unnecessary initialization
  - Extra indirections
  - Ignoring data locality
  - Excessive lock protection

- Cycle eaters multiply out of control

Classical Performance Analysis

1. Profile an execution
2. Inspect code regions with “high resource usage” (aka hotspots)
3. Improve code in these hotspots

Hotspot Analysis Is Insufficient

- It monitors metrics at a coarse granularity
  - Instruction Per Cycle (IPC), cache misses
  - Quantifies the average behavior over a time window
  - Never conveys any semantic meaning of an execution

- It cannot inform whether the hardware is being used *fruitfully*
  - `exp(const, const)` in a loop is wasteful use of FPU

- It may, in fact, mislead by acclaiming such loop with a high IPC
  - We have instances of lower IPC codes with a shorter running time

- Monitoring myriad PMU counters is **data rich** but **insight poor**

Focus on resource *wastage*
in addition to resource *usage*

From Resource Usage to Wastage

Look for prodigal resource consumption

- Wasteful data movement
  - Useless memory accesses [CGO’12]
    - Dead stores: stored value got overwritten without use
  - Redundant memory accesses [ASPLOS’17]
    - Redundant stores: write same values to a memory location
  - Unnecessary cacheline ping-ponging [PPoPP’18]
    - False sharing, contention

- Wasteful computation
  - Symbolically equivalent computation [PACT’15]
    - a=pow(b, c); d=pow(b, c);
  - Result equivalent computation [ASPLOS’17]
    - a=b*b-c*c; d=(b+c)*(b-c)

- Wasteful synchronization
  - Redundant barriers [PPoPP’15]

From Resource Usage to Wastage

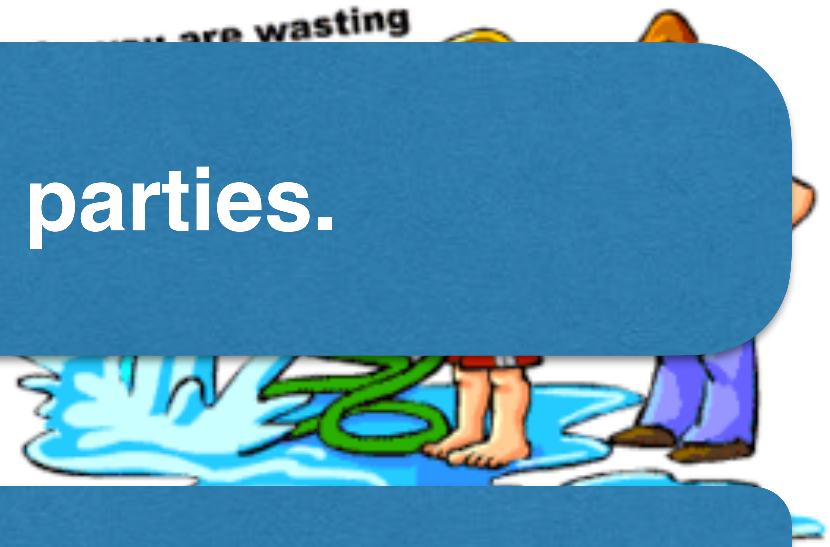
Typically involves two or more parties.

  * Dead stores: stored value got overwritten without use
- Redundant memory accesses [ASPLOS'17]
  * Redundant stores: write same values to a memory location
- Unnecessary cacheline ping-ponging [PPoPP'18]
  * False sharing, contention

- Wasteful computation
  - Symbolically equivalent computation [PACT'15]
    * a=pow(b, c); d=pow(b, c);
  - Result equivalent computation [ASPLOS'17]
    * a=b*b-c*c; d=(b+c)*(b-c)

- Wasteful synchronization
  - Redundant barriers [PPoPP'15]

From Resource Usage to Wastage



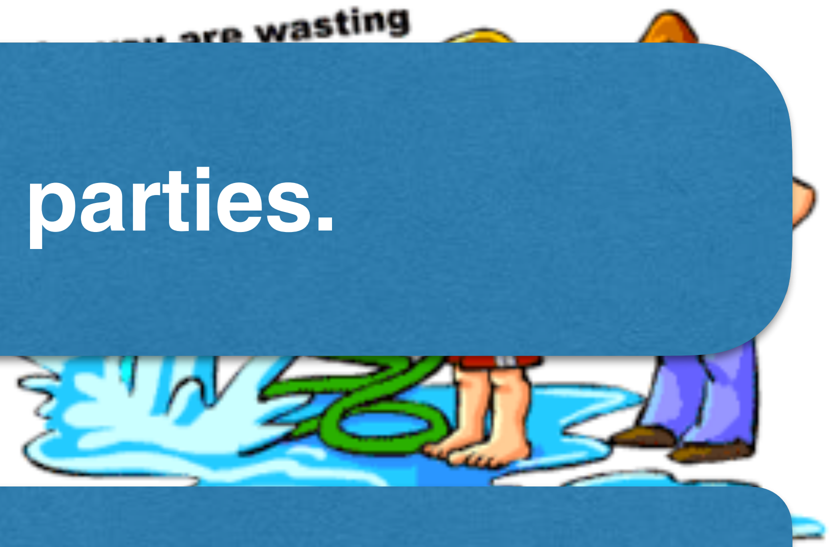
Typically involves two or more parties.

- Useless memory accesses [CGO'12]
    - Dead stores: stored value got overwritten without use
- Redundant memory accesses [ASPLOS'17]

Distinguishes useful from wasteful:
total memory accesses vs. useless memory accesses

- Wasteful computation
    - Symbolically equivalent computation [PACT'15]
        - a=pow(b, c); d=pow(b, c);
    - Result equivalent computation [ASPLOS'17]
        - a=b*b-c*c; d=(b+c)*(b-c)
- Wasteful synchronization
    - Redundant barriers [PPoPP'15]

From Resource Usage to Wastage



Typically involves two or more parties.

- Useless memory accesses [CGO'12]
  - Dead stores: stored value got overwritten without use
- Redundant memory accesses [ASPLOS'17]

Distinguishes useful from wasteful:
total memory accesses vs. useless memory accesses

- Wasteful computation
  - Symbolically equivalent computation [PACT'15]
    - a=pow(b,c); d=pow(b,c);

One step closer to reconstructing the semantic meaning (or lack there of) in an execution.

Dead Writes: Example

Riemann solver kernel
3-level nested loop
20% execution time

```
do k
 do j
  do i
   Wgdnv(i, j, k, 0) = …
   Wgdnv(i, j, k, inorm) = …
   Wgdnv(i, j, k, 4) = …
   if (spout.le.0.0d0) then
    Wgdnv(i, j, k, 0) = …
    Wgdnv(i, j, k, inorm) = …
    Wgdnv(i, j, k, 4) = …
   endif
   if (spin.gt.0.0d0) then
    Wgdnv(i, j, k, 0) = …
    Wgdnv(i, j, k, inorm) = …
    Wgdnv(i, j, k, 4) = …
   endif
```

- Chombo [LBNL]: AMR framework for solving PDEs
- Compilers can't eliminate all dead writes because of:
  - Aliasing / ambiguity
  - Aggregate variables
  - Function boundaries
  - Late binding
  - Partial deadness

Dead Writes: Example

Code lacked “design for performance”

```
do k
 do j
  do i
   Wgdnv(i, j, k, 0) = …
   Wgdnv(i, j, k, inorm) = …
   Wgdnv(i, j, k, 4) = …
   if (spout.le.0.0d0) then
    Wgdnv(i, j, k, 0) = …
    Wgdnv(i, j, k, inorm) = …
    Wgdnv(i, j, k, 4) = …
   endif
   if (spin.gt.0.0d0) then
    Wgdnv(i, j, k, 0) = …
    Wgdnv(i, j, k, inorm) = …
    Wgdnv(i, j, k, 4) = …
   endif
```

Better code: Use else-if nesting

```
do k
 do j
  do i
   if (spin.gt.0.0d0) then
    Wgdnv(i, j, k, 0) = …
    Wgdnv(i, j, k, inorm) = …
    Wgdnv(i, j, k, 4) = …
   elif (spout.le.0.0d0) then
    Wgdnv(i, j, k, 0) = …
    Wgdnv(i, j, k, inorm) = …
    Wgdnv(i, j, k, 4) = …
   else
    Wgdnv(i, j, k, 0) = …
    Wgdnv(i, j, k, inorm) = …
    Wgdnv(i, j, k, 4) = …
   endif
```

Dead Writes: Example

Code lacked “design for performance”

```
do k
 do j
  do i
   Wgdnv(i, j, k, 0) = …
   Wgdnv(i, j, k, inorm) = …
   Wgdnv(i, j, k, 4) = …
   if (spout.le.0.0d0) then
    Wgdnv(i, j, k, 0) = …
    Wgdnv(i, j, k, inorm) = …
    Wgdnv(i, j, k, 4) = …
   endif
   if (spin.gt.0.0d0) then
    Wgdnv(i, j, k, 0) = …
    Wgdnv(i, j, k, inorm) = …
    Wgdnv(i, j, k, 4) = …
   endif
```

Better code: Use else-if nesting

20% speedup of the loop

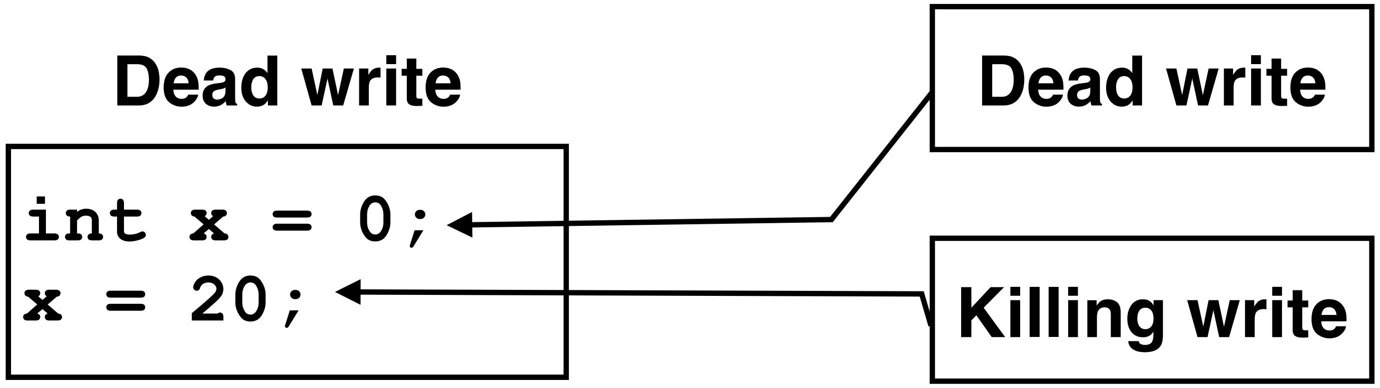
```
do k
 do j
  do i
   if (spin.gt.0.0d0) then
    Wgdnv(i, j, k, 0) = …
    Wgdnv(i, j, k, inorm) = …
    Wgdnv(i, j, k, 4) = …
   elif (spout.le.0.0d0) then
    Wgdnv(i, j, k, 0) = …
    Wgdnv(i, j, k, inorm) = …
    Wgdnv(i, j, k, 4) = …
   else
    Wgdnv(i, j, k, 0) = …
    Wgdnv(i, j, k, inorm) = …
    Wgdnv(i, j, k, 4) = …
   endif
```

Dead Writes

- Accessing memory is expensive on modern architectures
  - ✦ Multiple levels of hierarchy, cores share cache—>limited bandwidth per core
- Unnecessary writes
  - ✦ Cause unnecessary capacity miss and coherence traffic —> affects resource shared system
  - ✦ Wear out NVM-based or disk-based memory

Dead write: Two writes to the same memory location without an intervening read

Dead write



```
int x = 0;
x = 20;
```

Not a dead write

```
int x = 0;
Print(x);
x = 20;
```

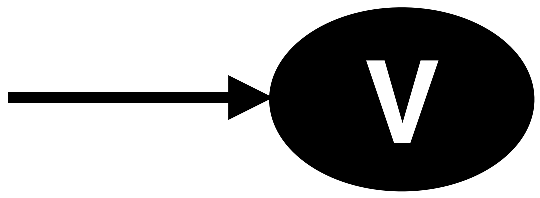
DeadSpy: Runtime Dead Write Detection

- Monitor every load and store in a program
- Maintain state information for each memory byte referenced by the program
- Detect every dead write in an execution with an automaton

[CGO’12] “DeadSpy: A Tool to Pinpoint Program Inefficiencies”

DeadSpy: Runtime Dead Write Detection

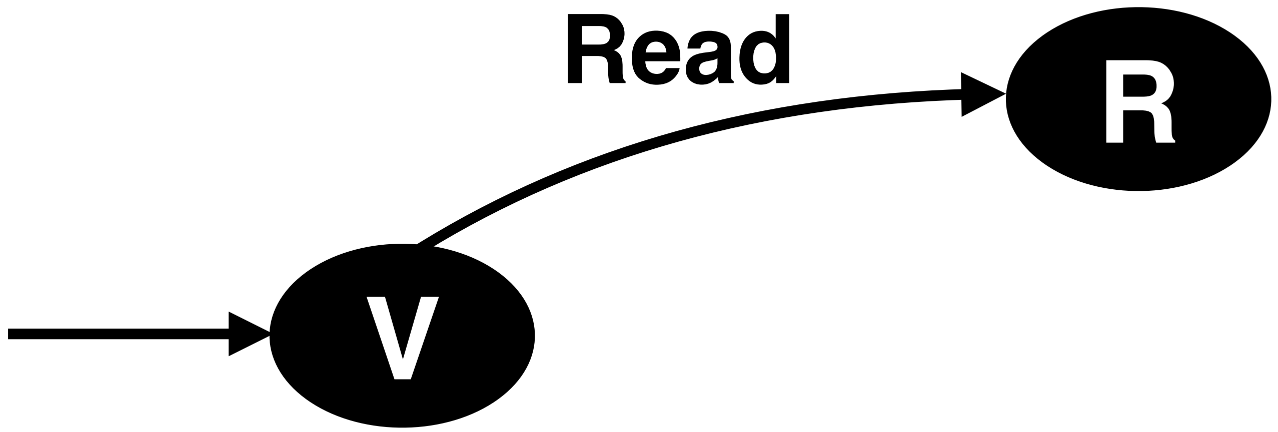
- Monitor every load and store in a program
- Maintain state information for each memory byte referenced by the program
- Detect every dead write in an execution with an automaton



[CGO'12] "DeadSpy: A Tool to Pinpoint Program Inefficiencies"

DeadSpy: Runtime Dead Write Detection

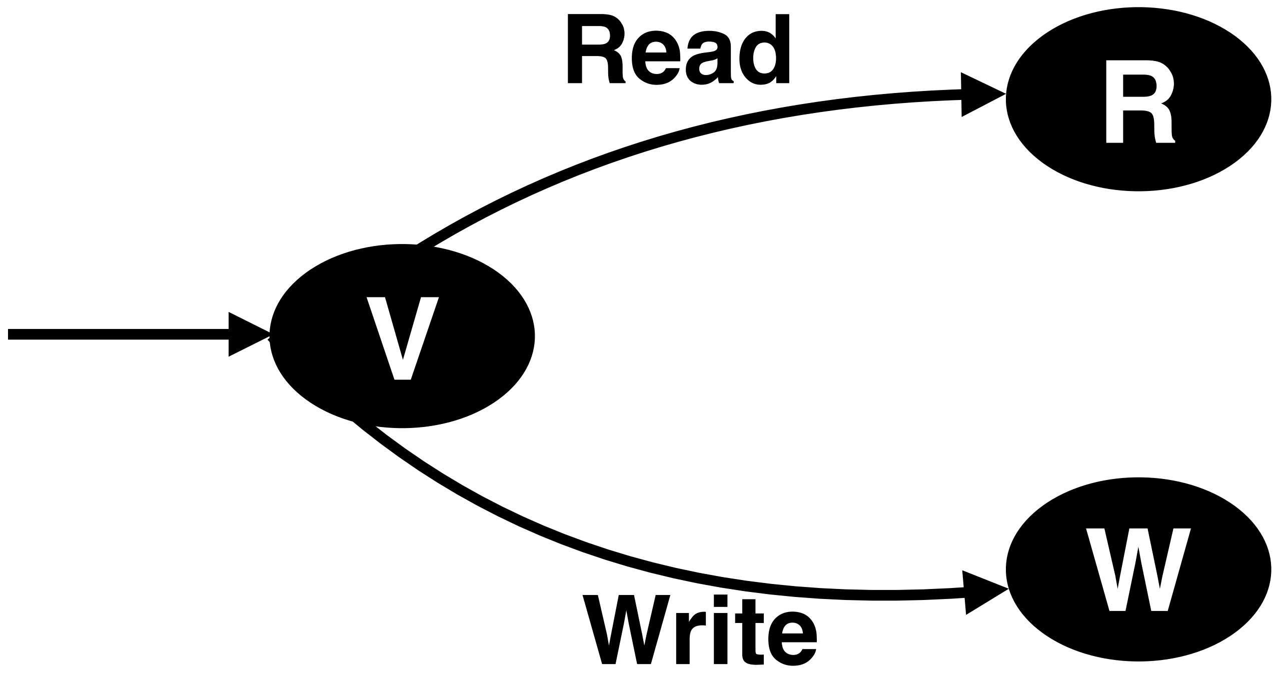
- Monitor every load and store in a program
- Maintain state information for each memory byte referenced by the program
- Detect every dead write in an execution with an automaton



[CGO’12] “DeadSpy: A Tool to Pinpoint Program Inefficiencies”

DeadSpy: Runtime Dead Write Detection

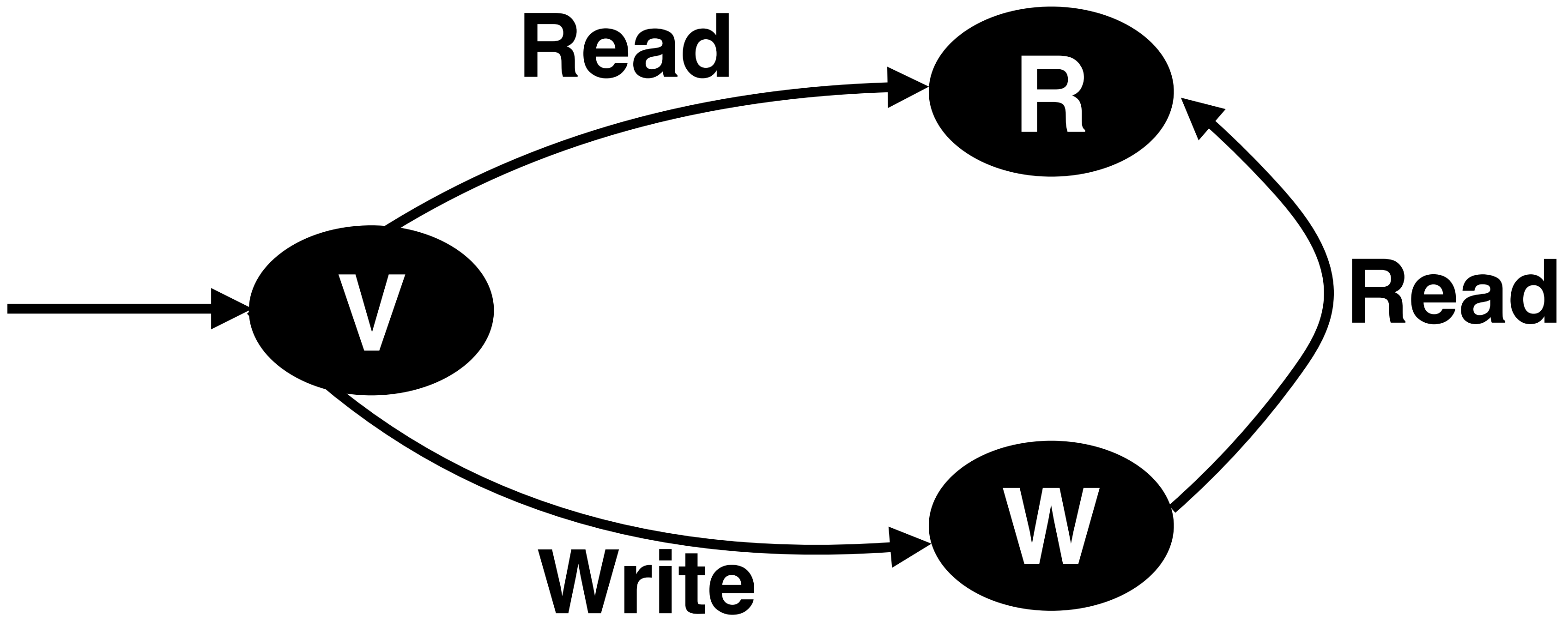
- Monitor every load and store in a program
- Maintain state information for each memory byte referenced by the program
- Detect every dead write in an execution with an automaton



[CGO'12] "DeadSpy: A Tool to Pinpoint Program Inefficiencies"

DeadSpy: Runtime Dead Write Detection

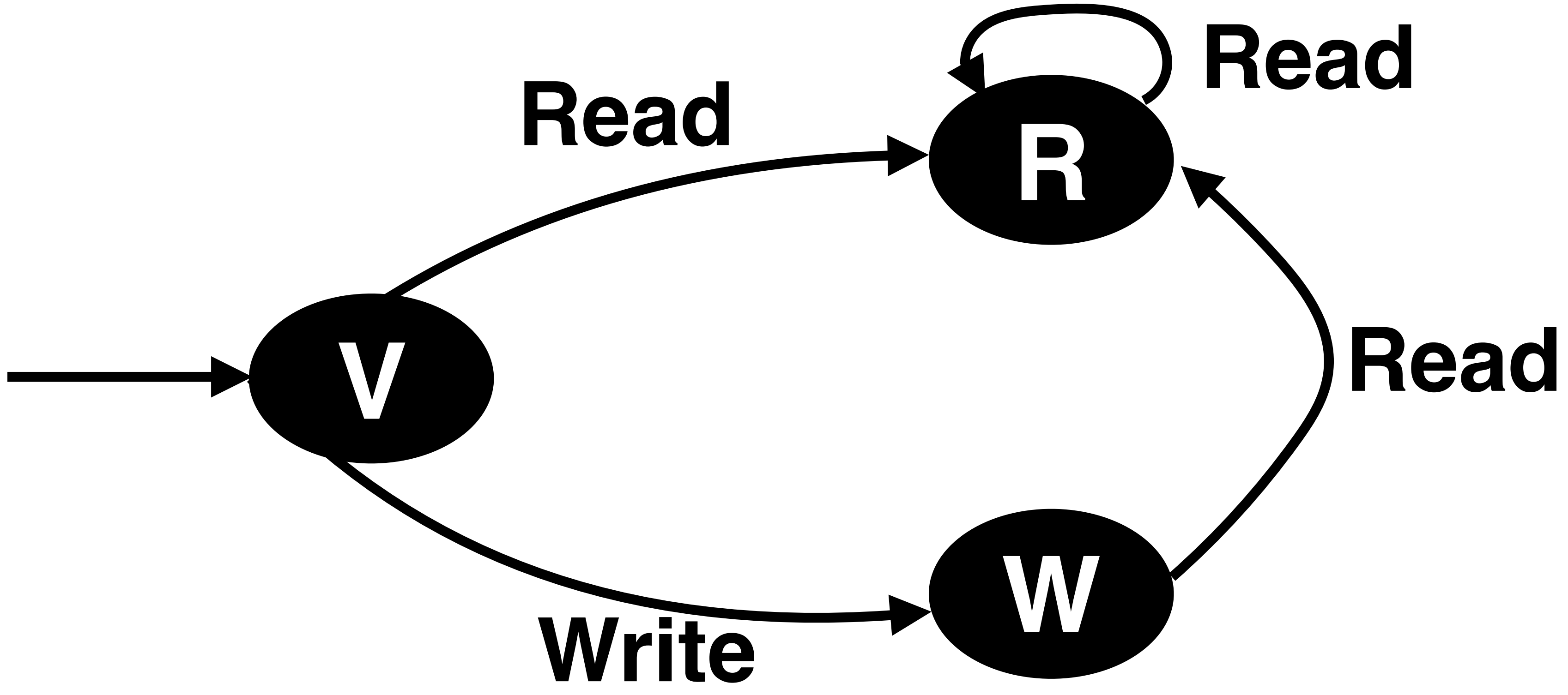
- Monitor every load and store in a program
- Maintain state information for each memory byte referenced by the program
- Detect every dead write in an execution with an automaton



[CGO'12] "DeadSpy: A Tool to Pinpoint Program Inefficiencies"

DeadSpy: Runtime Dead Write Detection

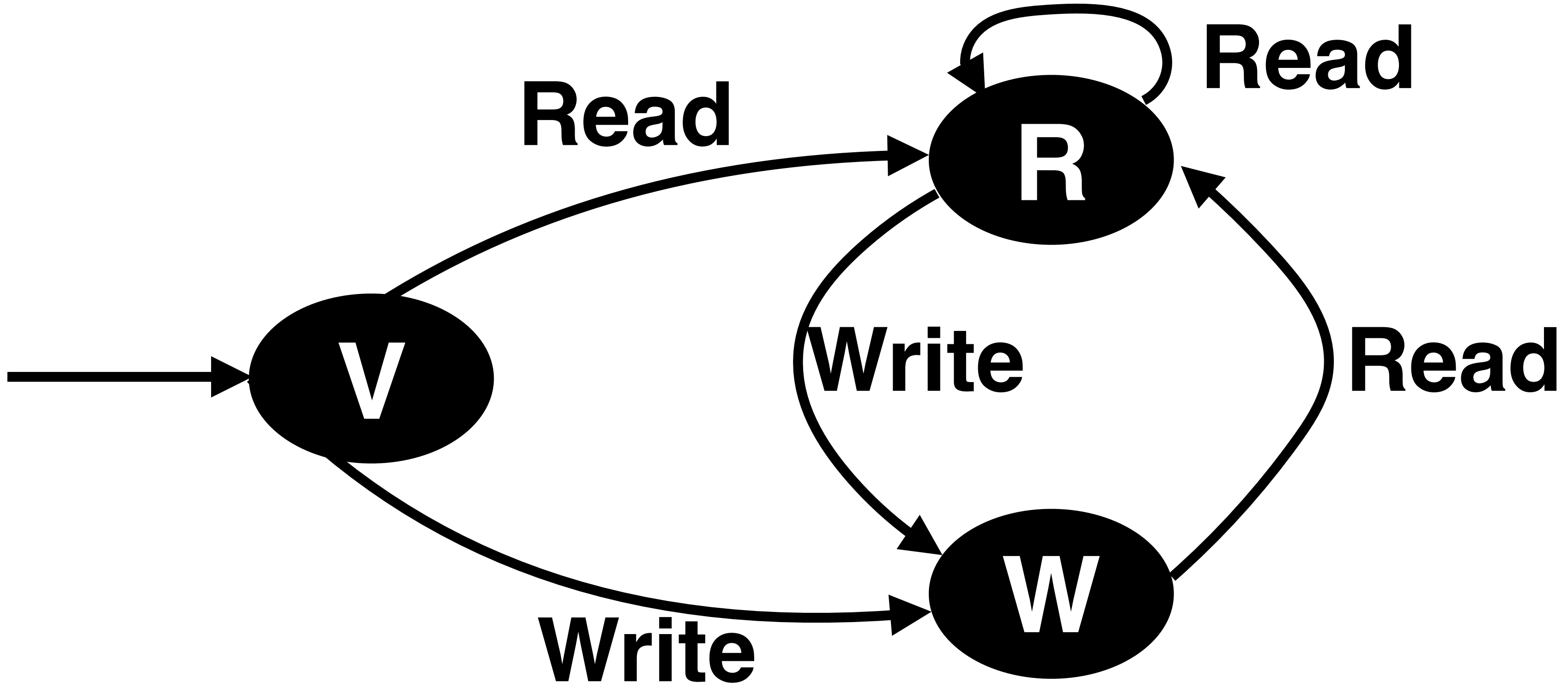
- Monitor every load and store in a program
- Maintain state information for each memory byte referenced by the program
- Detect every dead write in an execution with an automaton



[CGO'12] "DeadSpy: A Tool to Pinpoint Program Inefficiencies"

DeadSpy: Runtime Dead Write Detection

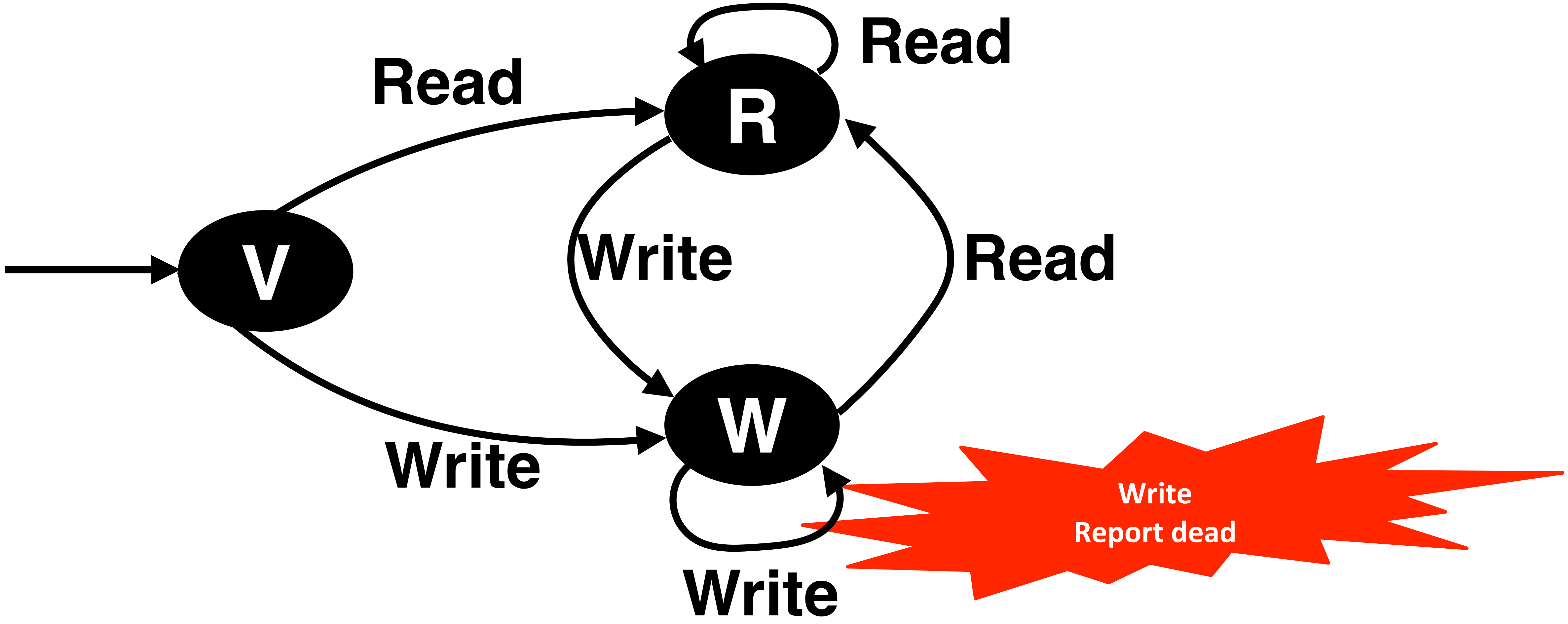
- Monitor every load and store in a program
- Maintain state information for each memory byte referenced by the program
- Detect every dead write in an execution with an automaton



[CGO'12] "DeadSpy: A Tool to Pinpoint Program Inefficiencies"

DeadSpy: Runtime Dead Write Detection

- Monitor every load and store in a program
- Maintain state information for each memory byte referenced by the program
- Detect every dead write in an execution with an automaton



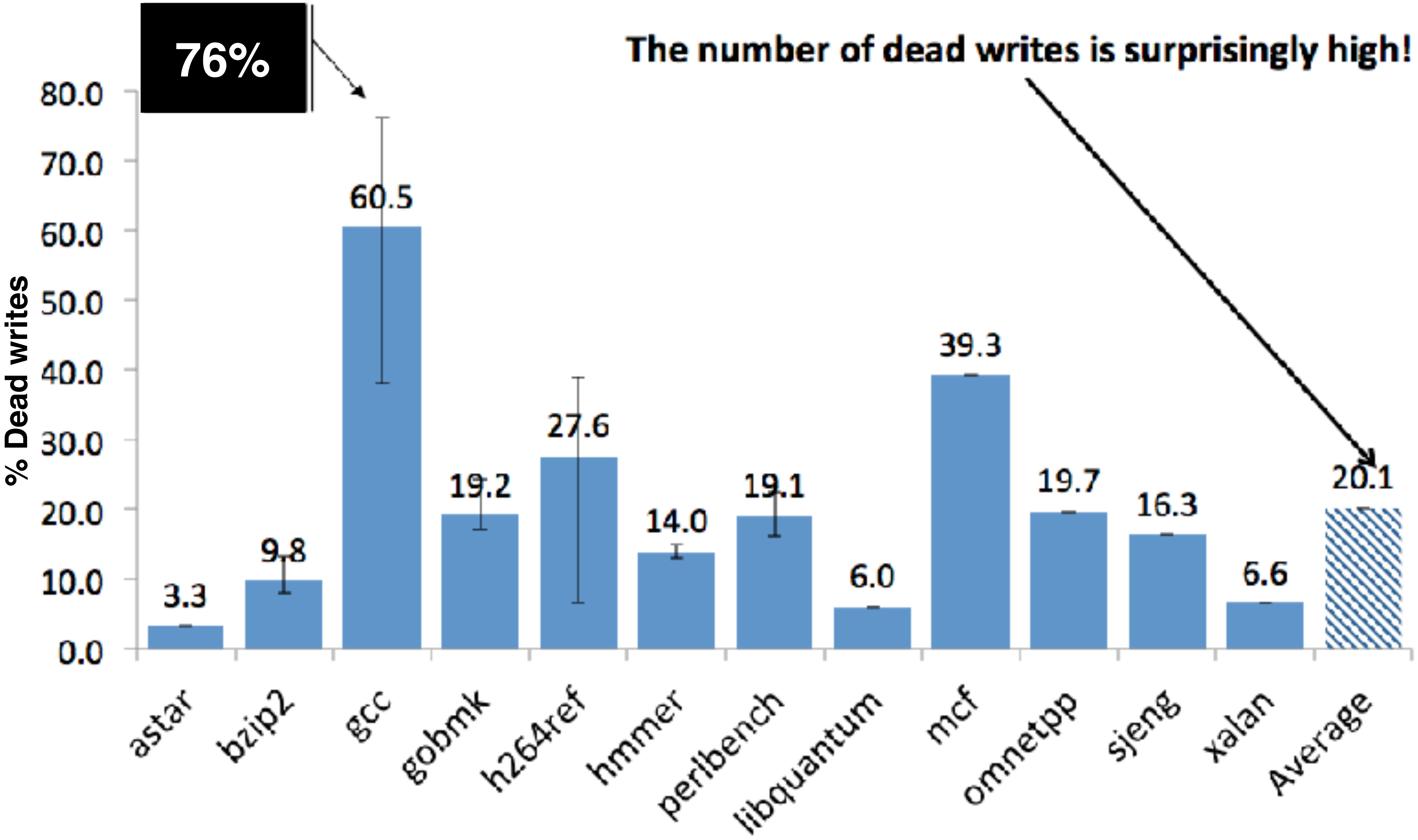
[CGO'12] "DeadSpy: A Tool to Pinpoint Program Inefficiencies"

Advantages

- No aliasing problems
  - Runtime monitoring at virtual address level
- No logical scope limitations; can detect across
  - Functions
  - Modules
  - Libraries
- No false positives or false negatives
  - Every reported *instance* is a dead write
- Disadvantage: input sensitive

Dead Writes in SPEC CPU2006

Lower is better



Across compilers and optimization levels

GCC: Use of Inappropriate Data Structure

```
static void loop_regs_scan (const struct loop *  loop, int  extra_size) {

 last_set = xcalloc (regs->num, sizeof (rtx));
  /* Scan the loop, recording register usage.  */
  for (Instrunction insn in loop)  {
     if (PATTERN (insn) sets a register)
        count_one_set (regs, insn, PATTERN (insn), last_set);
     if (Label(insn)) // new  BBL
        memset (last_set, 0, regs->num * sizeof (rtx));
  }
…
}
```

- **Basic blocks are short**
- **Median use: 2 unique elements**
- **<u>Dense array is a poor data structure choice</u>**

GCC: Use of Inappropriate Data Structure

Alloc and zero init 16,937 element (132KB) array

```
static void loop_                                           size) {

 last_set = xcalloc (regs->num, sizeof (rtx));
  /* Scan the loop, recording register usage.  */
  for (Instrunction insn in loop)  {
     if (PATTERN (insn) sets a register)
        count_one_set (regs, insn, PATTERN (insn), last_set);
     if (Label(insn)) // new  BBL
        memset (last_set, 0, regs->num * sizeof (rtx));
  }
…
}
```

- **Basic blocks are short**
- **Median use: 2 unique elements**
- **Dense array is a poor data structure choice**

GCC: Use of Inappropriate Data Structure

Alloc and zero init 16,937 element (132KB) array

```
static void loop_                                       size) {

 last_set = xcalloc (regs->num, sizeof (rtx));
  /* Scan the loop, recording register usage.  */
  for (Instrunction insn in loop)  {
     if (PATTERN (insn) sets a register)
        count_one_set (regs, insn, PATTERN (insn), last_set);
     if (Label(insn)) // new  BBL
        memset (last_set, 0, regs->num * sizeof (rtx));
  }
…
}
```

Dead

Killing

- **Basic blocks are short**
- **Median use: 2 unique elements**
- **Dense array is a poor data structure choice**

GCC: Use of Inappropriate Data Structure

Alloc and zero init 16,937 element (132KB) array

```
static void loop_                                         size) {

 last_set = xcalloc (regs->num, sizeof (rtx));
  /* Scan the loop, recording register usage.  */
  for (Instrunction insn in loop)  {
     if (PATTERN (insn) sets a register)
        count_one_set (regs, insn, PATTERN (insn), last_set);
     if (Label(insn)) // new  BBL
        memset (last_set, 0, regs->num * sizeof (rtx));
  }
…
}
```

Dead

Killing

Dead

Reinitializes 16,937 elements each time

- Basic blocks are short
- Median use: 2 unique elements
- <u>Dense array is a poor data structure choice</u>

GCC: Use of Inappropriate Data Structure

```
static void loop_r                                             size) {

 last_set = xcalloc (regs->num, sizeof (rtx));
  /* Scan the loop, recording register usage.  */
  for (Instrunction insn in loop)  {
     if (PATTERN (insn) sets a register)
        count_one_set (regs, insn, PATTERN (insn), last_set);
     if (Label(insn)) // new  BBL
        memset (last_set, 0, regs->num * sizeof (rtx));
  }
…
}
```

Alloc and zero init 16,937 element (132KB) array

Dead

Killing

Dead

Reinitializes 16,937 elements each time

- **Basic blocks are short**
- **Median use: 2 unique elements**
- **Dense array is a poor data structure choice**

Replaced array with a sparse data structure

> 28% running time improvement

Bzip2: Aggressive Optimization

```
Bool mainGtU ( UInt32  i1,  UInt32  i2,
UChar*  block,…) {
  Int32  k; UChar  c1, c2; UInt16 s1, s2;
   /* 1 */
  c1 = block[i1]; c2 = block[i2];
  if (c1 != c2) return (c1 > c2);
  i1++; i2++;
  /* 2 */
  c1 = block[i1]; c2 = block[i2];
  if (c1 != c2) return (c1 > c2);
  i1++; i2++;
  /* 3 */
  c1 = block[i1]; c2 = block[i2];

 /* 12 */
 c1 = block[i1]; c2 = block[i2];
  if (c1 != c2) return (c1 > c2);
  i1++; i2++;

  REST OF THE FUNCTION
```

Early Return

Bzip2: Aggressive Optimization

```
Bool mainGtU ( UInt32  i1,  UInt32  i2,
UChar*  block,…) {
  Int32  k; UChar  c1, c2; UInt16 s1, s2;
   /* 1 */
  c1 = block[i1]; c2 = block[i2];
  if (c1 != c2) return (c1 > c2);
  i1++; i2++;
  /* 2 */
  c1 = block[i1]; c2 = block[i2];
  if (c1 != c2) return (c1 > c2);
  i1++; i2++;
  /* 3 */
  c1 = block[i1]; c2 = block[i2];

 /* 12 */
 c1 = block[i1]; c2 = block[i2];
  if (c1 != c2) return (c1 > c2);
  i1++; i2++;

  REST OF THE FUNCTION
```

Early Return

```
Bool mainGtU ( UInt32  i1,  UInt32  i2, UChar*
block,…) {
  Int32  k; UChar  c1, c2; UInt16 s1, s2;
tmp1 = i1 +1;
tmp2 = i1 +2;
  …
tmp12 = i1 +12;
   /* 1 */
  c1 = block[i1]; c2 = block[i2];
  if (c1 != c2) return (c1 > c2);
  ~~i1++;~~ i2++;
  /* 2 */
  c1 = block[tmp1]; c2 = block[i2];
  if (c1 != c2) return (c1 > c2);
  ~~i1++;~~ i2++;
  /* 3 */
  c1 = block[tmp2]; c2 = block[i2];

  /* 12 */
  c1 = block[tmp11]; c2 = block[i2];
  if (c1 != c2) return (c1 > c2);
  ~~i1++;~~ i2++;

  REST OF THE FUNCTION
```

Early Return

Bzip2: Aggressive Optimization

```
Bool mainGtU ( UInt32  i1,  UInt32  i2,
UChar*  block,...) {
  Int32  k; UChar  c1, c2; UInt16 s1, s2;
  /* 1 */
  c1 = block[i1]; c2 = block[i2];
  if (c1 != c2) return (c1 > c2);
  i1++; i2++;
  /* 2 */
  c1 = block[i1]; c2 = block[i2];
  if (c1 != c2) return (c1 > c2);
  i1++; i2++;
  /* 3 */
  c1 = block[i1]; c2 = block[i2];

 /* 12 */
 c1 = block[i1]; c2 = block[i2];
  if (c1 != c2) return (c1 > c2);
  i1++; i2++;

  REST OF THE FUNCTION
```

Early Return

```
Bool mainGtU ( UInt32  i1         2  i2, UChar*
block,...) {
  Int32  k; UCh          c2; UInt16 s1, s2;
tmp1 = i1 +1;
tmp2 = i1 +2;
  …
tmp12 = i1 +12;
   /* 1 */
  c1 = block[i1]; c2 = block[
  if (c1 != c2) return (c1 > c2);
  ~~i1++;~~ i2++;
  /* 2 */
  c1 = block[tmp1]; c2 = block[i2];
  if (c1 != c2) return (c1 > c2);
  ~~i1++;~~ i2++;
  /* 3 */
  c1 = block[tmp2]; c2 = block[i2];

 /* 12 */
 c1 = block[tmp11]; c2 = block[i2];
  if (c1 != c2) return (c1 > c2);
  ~~i1++;~~ i2++;

  REST OF THE FUNCTION
```

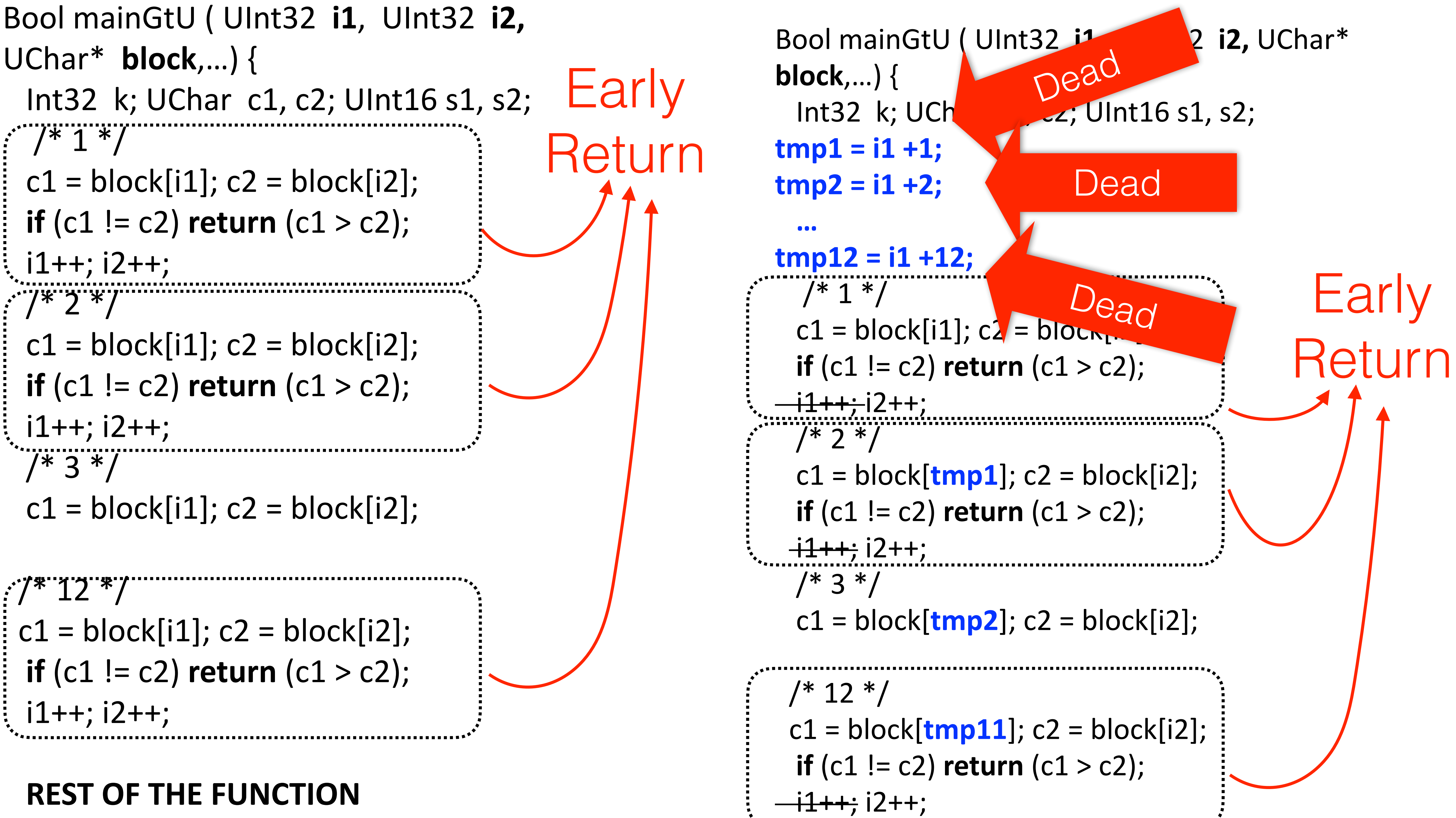
Dead

Dead

Dead

Early Return

Bzip2: Aggressive Optimization



```
Bool mainGtU ( UInt32  i1,  UInt32  i2,
UChar*  block,...) {
  Int32  k; UChar  c1, c2; UInt16 s1, s2;
  /* 1 */
  c1 = block[i1]; c2 = block[i2];
  if (c1 != c2) return (c1 > c2);
  i1++; i2++;
  /* 2 */
  c1 = block[i1]; c2 = block[i2];
  if (c1 != c2) return (c1 > c2);
  i1++; i2++;
  /* 3 */
  c1 = block[i1]; c2 = block[i2];

  /* 12 */
  c1 = block[i1]; c2 = block[i2];
  if (c1 != c2) return (c1 > c2);
  i1++; i2++;

  REST OF THE FUNCTION
```

```
Bool mainGtU ( UInt32  i1              2  i2, UChar*
block,...) {
  Int32  k; UCh              c2; UInt16 s1, s2;
tmp1 = i1 +1;
tmp2 = i1 +2;
  …
tmp12 = i1 +12;
  /* 1 */
  c1 = block[i1]; c2 = block[
  if (c1 != c2) return (c1 > c2);
  ~~i1++;~~ i2++;
  /* 2 */
  c1 = block[tmp1]; c2 = block[i2];
  if (c1 != c2) return (c1 > c2);
  ~~i1++;~~ i2++;
  /* 3 */
  c1 = block[tmp2]; c2 = block[i2];

  /* 12 */
  c1 = block[tmp11]; c2 = block[i2];
  if (c1 != c2) return (c1 > c2);
  ~~i1++;~~ i2++;
```

> 15% running time improvement

HMMER: Optimization Disabling Code Shape

Source code

```
for (i = 1; i <= L; i++) {
   for (k = 1; k <= M; k++) {
      ...
      ic[k] = mpp[k] + tpmi[k];
      if ((sc = ip[k] + tpii[k]) > ic[k])
         ic[k] = sc;
```

Machine-code@[-O3]

```
for (i = 1; i <= L; i++) {
   for (k = 1; k <= M; k++) {
      ...
      R1= mpp[k] + tpmi[k];
      ic[k] = R1;
      if ((sc = ip[k] + tpii[k]) > R1)
         ic[k] = sc;
```

HMMER: Optimization Disabling Code Shape

Source code

```
for (i = 1; i <= L; i++) {
  for (k = 1; k <= M; k++) {
     ...
     ic[k] = mpp[k] + tpmi[k];
     if ((sc = ip[k] + tpii[k]) > ic[k])
        ic[k] = sc;
```

Machine-code@[-O3]

```
for (i = 1; i <= L; i++) {
  for (k = 1; k <= M; k++) {
     ...
     R1= mpp[k] + tpmi[k];
     ic[k] = R1;                  Dead
     if ((sc = ip[k] + tpii[k]) > R1)
        ic[k] = sc;               Killing
```

HMMER: Optimization Disabling Code Shape

Source code

```
for (i = 1; i <= L; i++) {
   for (k = 1; k <= M; k++) {
       …
       ic[k] = mpp[k] + tpmi[k];
       if ((sc = ip[k] + tpii[k]) > ic[k])
          ic[k] = sc;
```

Machine-code@[-O3]

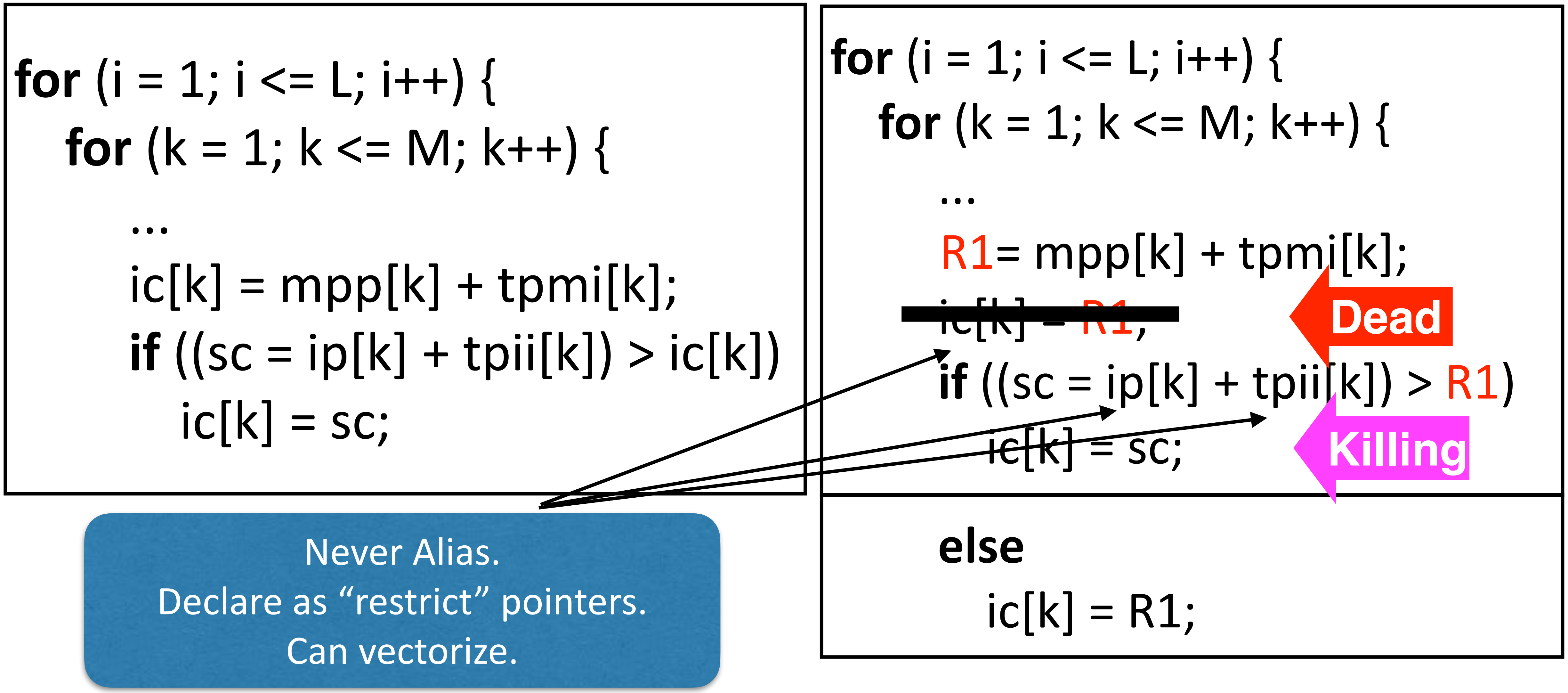
```
for (i = 1; i <= L; i++) {
   for (k = 1; k <= M; k++) {
       …
       R1= mpp[k] + tpmi[k];
       ~~ic[k] = R1;~~
       if ((sc = ip[k] + tpii[k]) > R1)
          ic[k] = sc;
```

Dead

Killing

```
       else
          ic[k] = R1;
```

HMMER: Optimization Disabling Code Shape



Source code

```
for (i = 1; i <= L; i++) {
  for (k = 1; k <= M; k++) {
    ...
    ic[k] = mpp[k] + tpmi[k];
    if ((sc = ip[k] + tpii[k]) > ic[k])
      ic[k] = sc;
```

Never Alias.
Declare as “restrict” pointers.
Can vectorize.

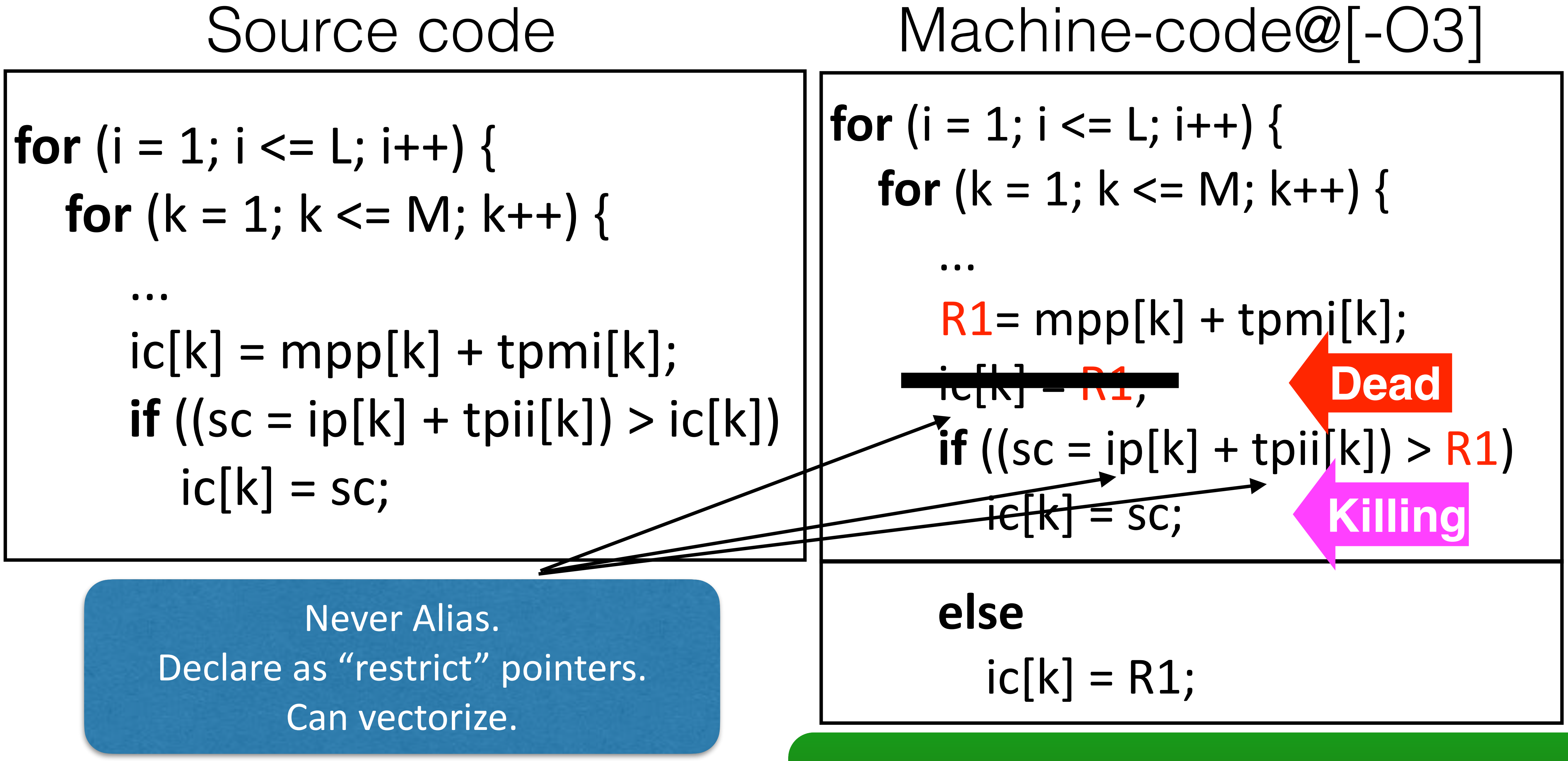
Machine-code@[-O3]

```
for (i = 1; i <= L; i++) {
  for (k = 1; k <= M; k++) {
    ...
    R1= mpp[k] + tpmi[k];
```

~~ic[k] = R1;~~

```
    if ((sc = ip[k] + tpii[k]) > R1)
      ic[k] = sc;

    else
      ic[k] = R1;
```

HMMER: Optimization Disabling Code Shape



Source code

```
for (i = 1; i <= L; i++) {
   for (k = 1; k <= M; k++) {
       ...
       ic[k] = mpp[k] + tpmi[k];
       if ((sc = ip[k] + tpii[k]) > ic[k])
          ic[k] = sc;
```

Never Alias.
Declare as “restrict” pointers.
Can vectorize.

Machine-code@[-O3]

```
for (i = 1; i <= L; i++) {
   for (k = 1; k <= M; k++) {
       ...
       R1= mpp[k] + tpmi[k];
       ~~ic[k] = R1;~~
       if ((sc = ip[k] + tpii[k]) > R1)
          ic[k] = sc;
       else
          ic[k] = R1;
```

> 16% running time improvement
> 40% with vectorization

Dead writes are surprisingly common even in fully optimized code. Algorithmic defects often show up as dead writes.

Computational Redundancies

Static analysis for value numbering

$x = a \oplus b$

$y = a \oplus b$

Can detect

Computational Redundancies

Static analysis for value numbering

$x^{\#3} = a^{\#1} \oplus b^{\#2} \longrightarrow$ <⊕, #1, #2> : #3 —> x

$y \quad = a^{\#1} \oplus b^{\#2}$

Can detect

Computational Redundancies

Static analysis for value numbering

$x^{\#3} = a^{\#1} \oplus b^{\#2}$

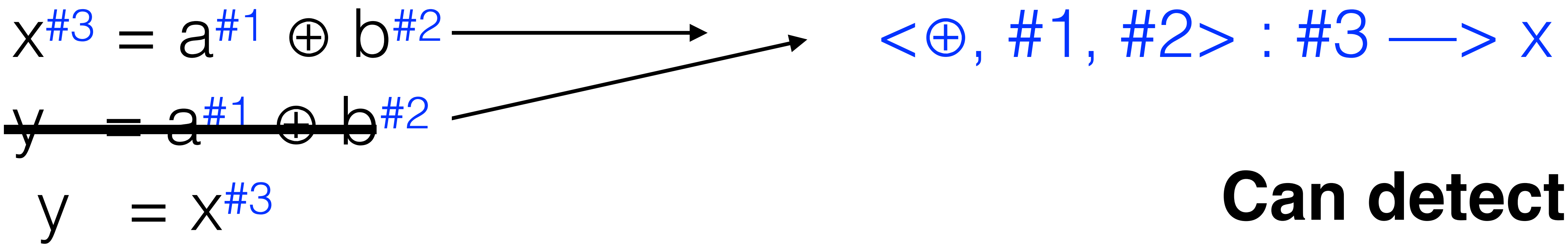
$y \quad = a^{\#1} \oplus b^{\#2}$

$\langle \oplus, \#1, \#2 \rangle : \#3 \longrightarrow x$

Can detect

Computational Redundancies

Static analysis for value numbering



$x^{\#3} = a^{\#1} \oplus b^{\#2}$

~~$y = a^{\#1} \oplus b^{\#2}$~~

$y = x^{\#3}$

$<\oplus, \#1, \#2> : \#3 \longrightarrow x$

Can detect

Computational Redundancies

Static analysis for value numbering

$x^{\#3} = a^{\#1} \oplus b^{\#2}$ → <⊕, #1, #2> : #3 —> x

~~$y = a^{\#1} \oplus b^{\#2}$~~ →

$y = x^{\#3}$

Can detect

```
FuncA(int &a, int &b, int &c, int &d) {
  x = a ⊕ b;
  y = c ⊕ d;
}
FuncB() {
  FuncA(m, n, m, n); // Invocation
}
```

May not detect

Optimization scope
and aliasing

RVN: Runtime Value Numbering

- Assign value numbers at runtime
- Propagate value numbers throughout execution
  - ✦ Use shadow memory and registers
- On each computation, check for the existence of a prior symbolically equivalent computation

[PACT'15] "Runtime Value Numbering: A Profiling Technique to Pinpoint Redundant Computations"

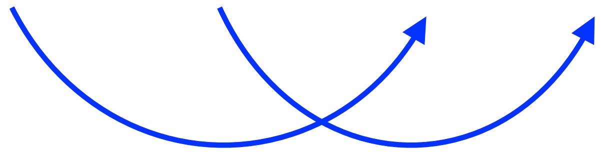
Runtime Value Numbering

```
FuncA(int &a   , int &b   , int &c   , int &d   ) {
  x   = a   ⊕  b   ;
  y   = c   ⊕  d   ;
}
FuncB() {
 FuncA(m#1, n#2, m   , n   ); // Invocation
}
```

m : #1
n : #2

Runtime Value Numbering

```
FuncA(int &a   , int &b   , int &c   , int &d   ) {
  x   = a   ⊕  b   ;
  y   = c   ⊕  d   ;
}
FuncB() {
  FuncA(m#1, n#2, m#1, n#2); // Invocation
}
```



m : #1
n : #2

Runtime Value Numbering

```
FuncA(int &a#1, int &b#2, int &c#1, int &d#2) {
  x    = a    ⊕  b    ;
  y    = c    ⊕  d    ;
}
FuncB() {
  FuncA(m#1, n#2, m#1, n#2); // Invocation
}
```

m : #1

n : #2

Runtime Value Numbering

FuncA(int &$a^{\#1}$, int &$b^{\#2}$, int &$c^{\#1}$, int &$d^{\#2}$) {
  $x^{\#3} = a^{\#1} \oplus b^{\#2}$;
  $y = c \oplus d$;
}
FuncB() {
  FuncA($m^{\#1}$, $n^{\#2}$, $m^{\#1}$, $n^{\#2}$); // Invocation
}

m : #1
n : #2
<⊕, #1, #2> : #3

Runtime Value Numbering

FuncA(int &$a^{\#1}$, int &$b^{\#2}$, int &$c^{\#1}$, int &$d^{\#2}$) {

  $x^{\#3} = a^{\#1} \oplus b^{\#2}$;

  $y \quad = c^{\#1} \oplus d^{\#2}$;

}

FuncB() {

  FuncA($m^{\#1}$, $n^{\#2}$, $m^{\#1}$, $n^{\#2}$); // Invocation

}

m : #1

n : #2

<⊕, #1, #2> : #3

Runtime Value Numbering

FuncA(int $\&a^{\#1}$, int $\&b^{\#2}$, int $\&c^{\#1}$, int $\&d^{\#2}$) {

$x^{\#3} = a^{\#1} \oplus b^{\#2}$;

$y = c^{\#1} \oplus d^{\#2}$;

}

FuncB() {

FuncA($m^{\#1}$, $n^{\#2}$, $m^{\#1}$, $n^{\#2}$); // Invocation

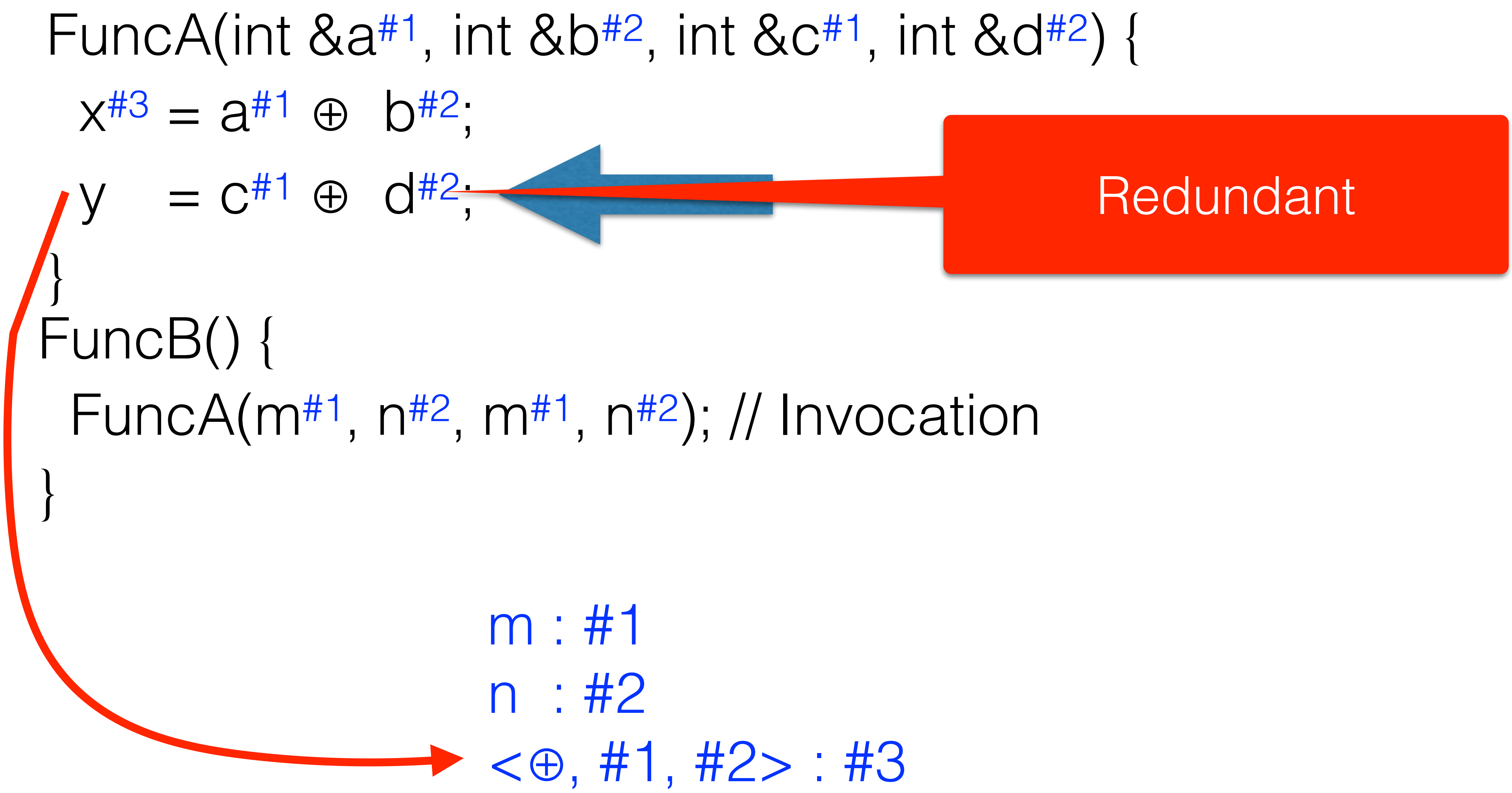
}

m : #1

n : #2

<⊕, #1, #2> : #3

Runtime Value Numbering



FuncA(int &$a^{\#1}$, int &$b^{\#2}$, int &$c^{\#1}$, int &$d^{\#2}$) {

$x^{\#3} = a^{\#1} \oplus b^{\#2}$;

$y = c^{\#1} \oplus d^{\#2}$;

}

FuncB() {

FuncA($m^{\#1}$, $n^{\#2}$, $m^{\#1}$, $n^{\#2}$); // Invocation

}

m : #1

n : #2

<⊕, #1, #2> : #3

Redundant Computations in BWaves

```
do k=1, nz
  km1 = mod(k+nz-2, nz) +1
  kp1 = mod(k, nz) + 1
  do j=1, ny
    jm1 = mod(j+ny-2, ny) + 1
    jp1 = mod(j, ny) + 1
    do i = 1, nx
      im1 = mod(i+nx-2, nx) + 1
      ip1 = mod(i, nx) +1
      …
    enddo
  enddo
enddo
```

Redundant Computations in BWaves

```
do k=1, nz
  km1 = mod(k+nz-2, nz) +1
  kp1 = mod(k, nz) + 1
  do j=1, ny
    jm1 = mod(j+ny-2, ny) + 1
    jp1 = mod(j, ny) + 1
    do i = 1, nx
      im1 = mod(i+nx-2, nx) + 1
      ip1 = mod(i, nx) +1
      …
    enddo
  enddo
enddo
```

Redundant computation
Loop k invariant

Redundant computation
Loop k, j invariant

Redundant Computations in BWaves

```
do k=1, nz
  km1 = mod(k+nz-2, nz) +1
  kp1 = mod(k, nz) + 1
  do j=1, ny
    jm1 = mod(j+ny-2, ny) + 1
    jp1 = mod(j, ny) + 1
    do i = 1, nx
      im1 = mod(i+nx-2, nx) + 1
      ip1 = mod(i, nx) +1
      …
    enddo
  enddo
enddo
```

Redundant computation
Loop k invariant

Redundant computation
Loop k, j invariant

i	**1**	2	3	4	…	98	99	**100**
im1	**100**	1	2	3	…	97	98	**99**
ip1	**2**	3	4	5	…	99	100	**1**

im1 = i -1
ip1 = i+1

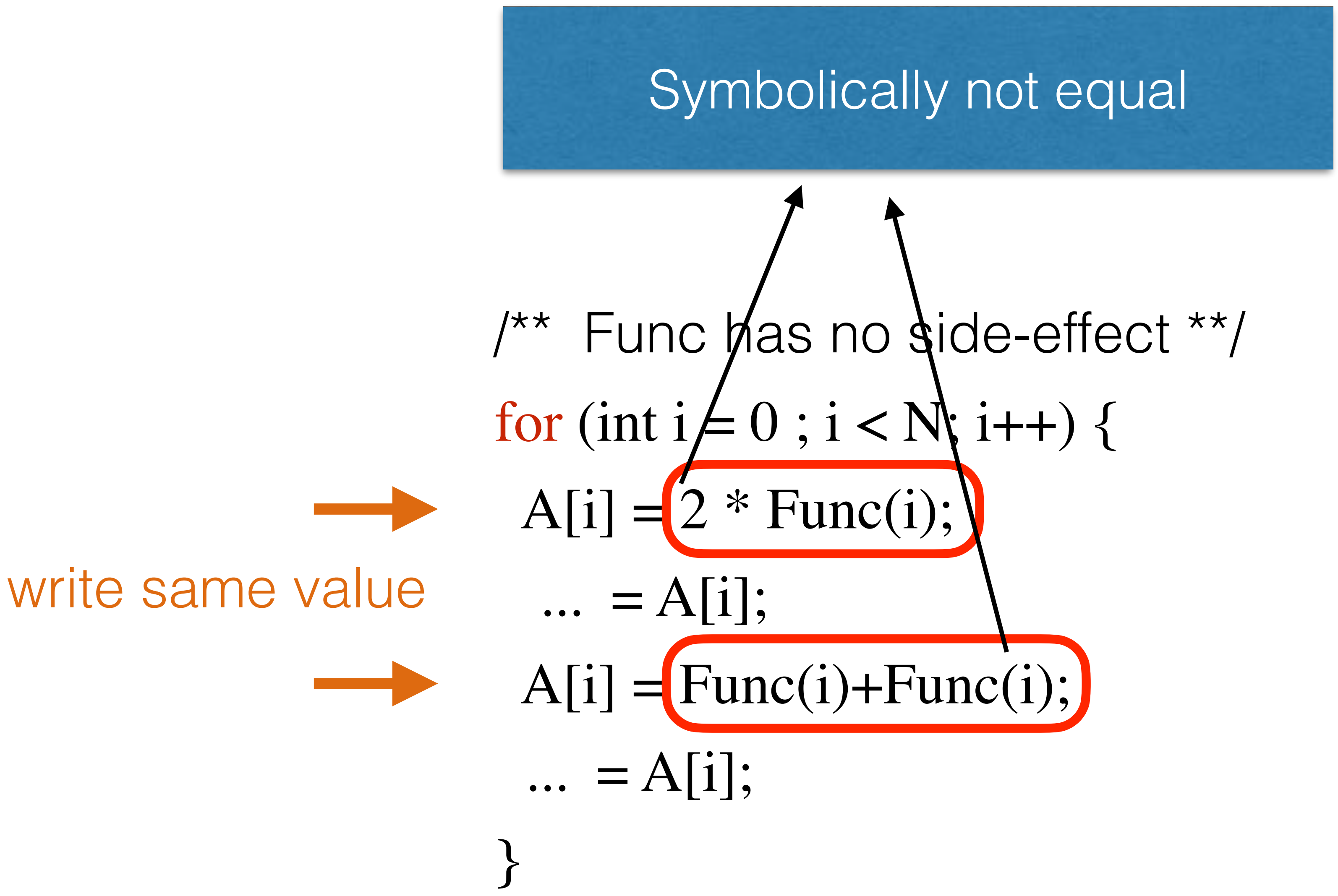
Loop peeling => 20% speedup

More Redundancies: Silent Stores

```
/**  Func has no side-effect **/
for (int i = 0 ; i < N; i++) {
  A[i] = 2 * Func(i);
   ...  = A[i];
  A[i] = Func(i)+Func(i);
  ...  = A[i];
}
```

write same value

More Redundancies: Silent Stores



Symbolically not equal

write same value

```
/**  Func has no side-effect **/
for (int i = 0 ; i < N; i++) {
  A[i] = 2 * Func(i);
   ...  = A[i];
  A[i] = Func(i)+Func(i);
   ...  = A[i];
}
```

More Redundancies: Silent Stores

Symbolically not equal

```
/**  Func has no side-effect **/
for (int i = 0 ; i < N; i++) {
  A[i] = 2 * Func(i);
   ...  = A[i];
  A[i] = Func(i)+Func(i);
  ...  = A[i];
}
```

write same value

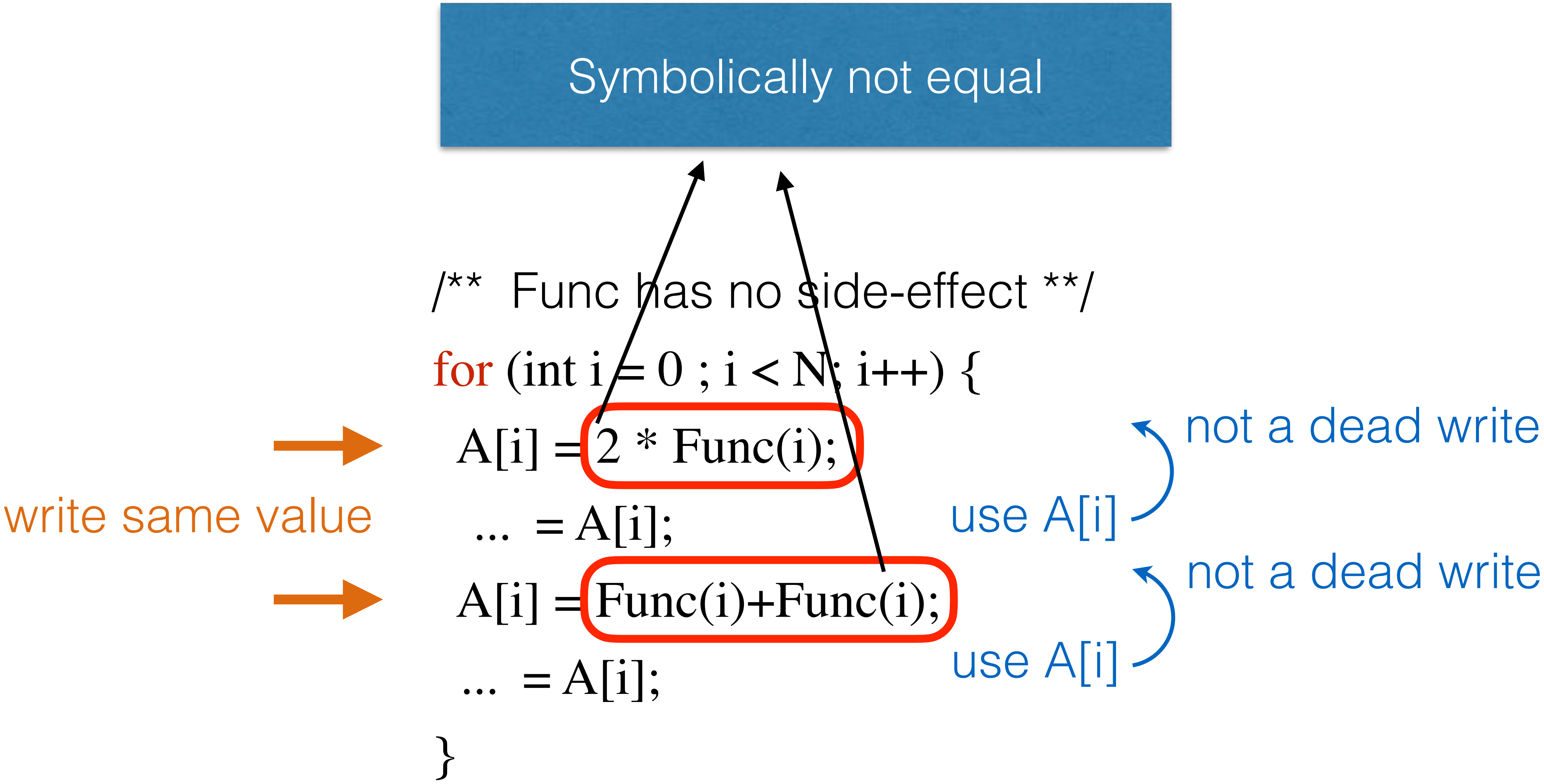
not a dead write

use A[i]

not a dead write

use A[i]

More Redundancies: Silent Stores



Symbolically not equal

```
/**  Func has no side-effect **/
for (int i = 0 ; i < N; i++) {
  A[i] = 2 * Func(i);
   ...  = A[i];
  A[i] = Func(i)+Func(i);
  ...  = A[i];
}
```

write same value

not a dead write

use A[i]

not a dead write

use A[i]

DeadSpy and RVN cannot detect this redundancy

RedSpy: From Value-Agnostic to Value-Aware

- Inspect **“runtime value”** produced by each operation

- Flag redundant if the same value overwrites the previous value (in registers or memory)

- DeadSpy: Value agnostic
  - Did not inspect the value at a location
  - Inspected the operation (read/write) on a location

[ASPLOS’17] “RedSpy: Exploring Value Locality in Software” ASPLOS Highlight Paper

Value Redundancy Types

- Temporal value redundancy
  - Same value overwrites the same storage location

```
x = 42;
…
x = 42;
```

- Spatial value redundancy
  - Nearby storage locations share a common value

```
a[100] = 42;
…
a[101] = 42;
```

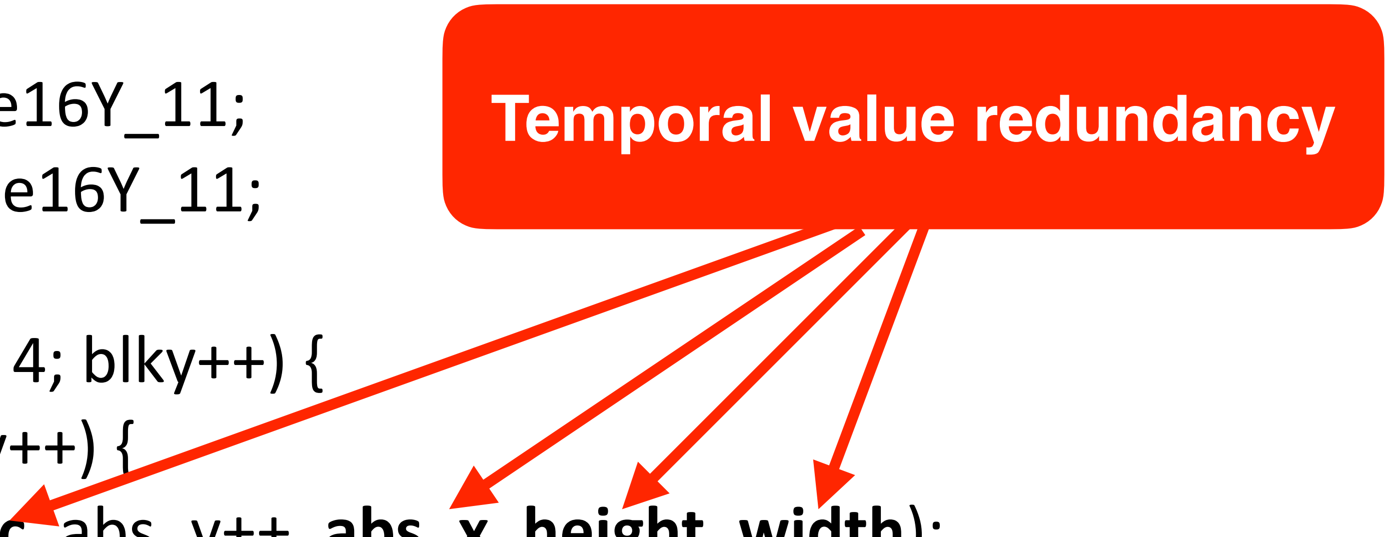
- Approximate value redundancy (temporal or spatial)

```
x = 42.0;
…
x = 42.01;
```

[ASPLOS'17] "RedSpy: Exploring Value Locality in Software" **ASPLOS Highlight Paper**

Value Redundancy: H264 Missed Inlining

```
for (…) {

  if(…)  fptr = FastLine16Y_11;
  else  fptr = UMVLine16Y_11;

  for (blky = 0; blky < 4; blky++) {
    for (y = 0; y < 4; y++) {
      retVal = fptr(pic, abs_y++, abs_x, height, width);
    }
  }
}
```

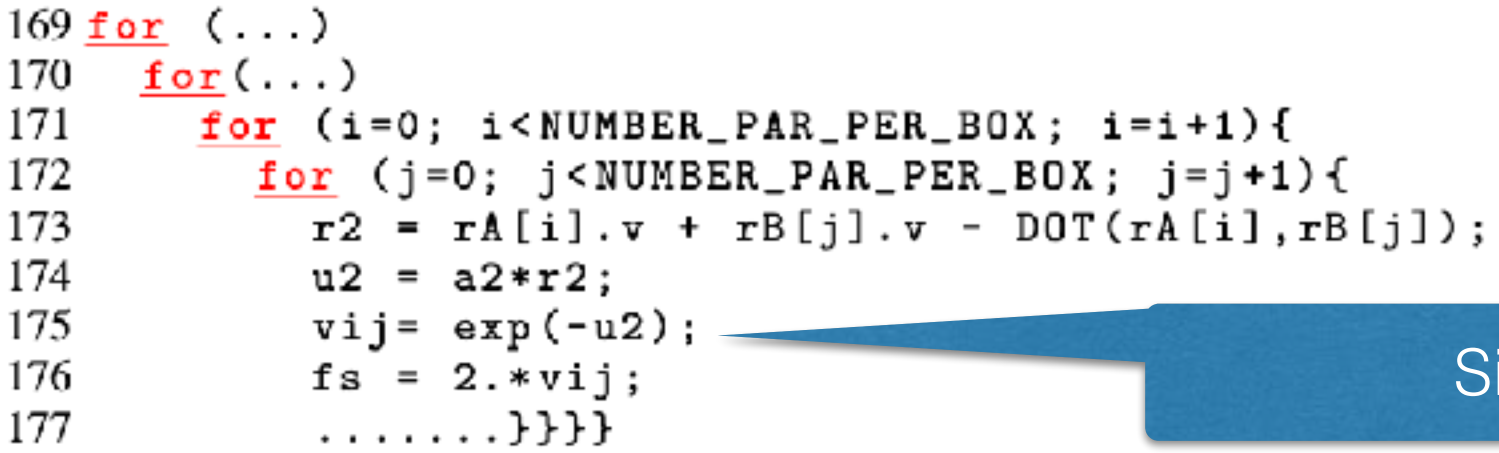


Temporal value redundancy

- Function not inlined: cross module and function pointer
- LTO+PGO inlines but adds a condition check in the loop
- Inlining + loop cloning/specialization —> 1.28x speedup

Temporal Redundancy in Rhodenia LavaMD

- OpenMP MD code: computes potential and relocation between particles in 3D space
- 90% time in the following loop
- > 60% silent stores in the loop



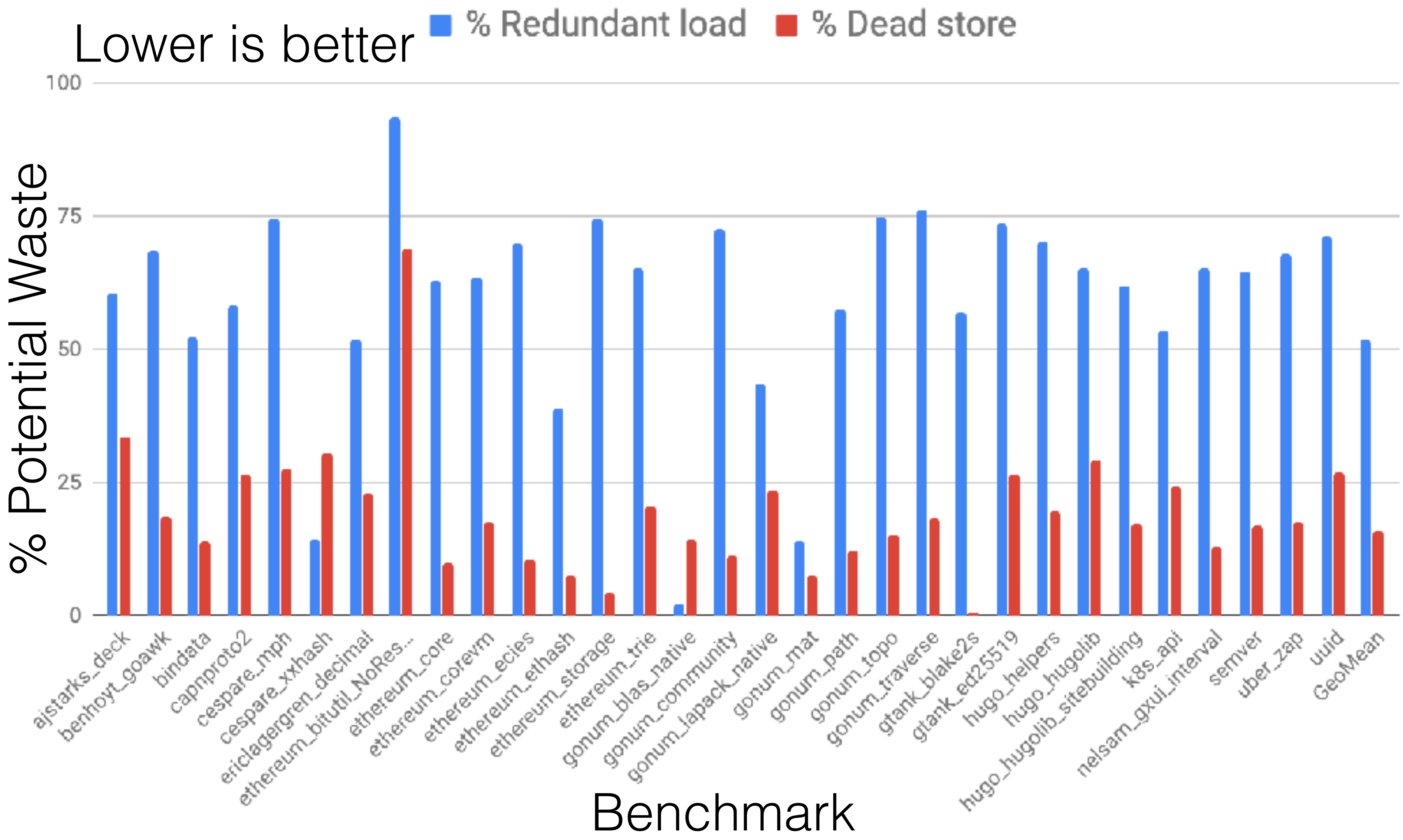
```
for  (...)
  for(...)
    for (i=0; i<NUMBER_PAR_PER_BOX; i=i+1){
      for (j=0; j<NUMBER_PAR_PER_BOX; j=j+1){
        r2 = rA[i].v + rB[j].v - DOT(rA[i],rB[j]);
        u2 = a2*r2;
        vij= exp(-u2);
        fs = 2.*vij;
        .......}}}}
```

Silent store

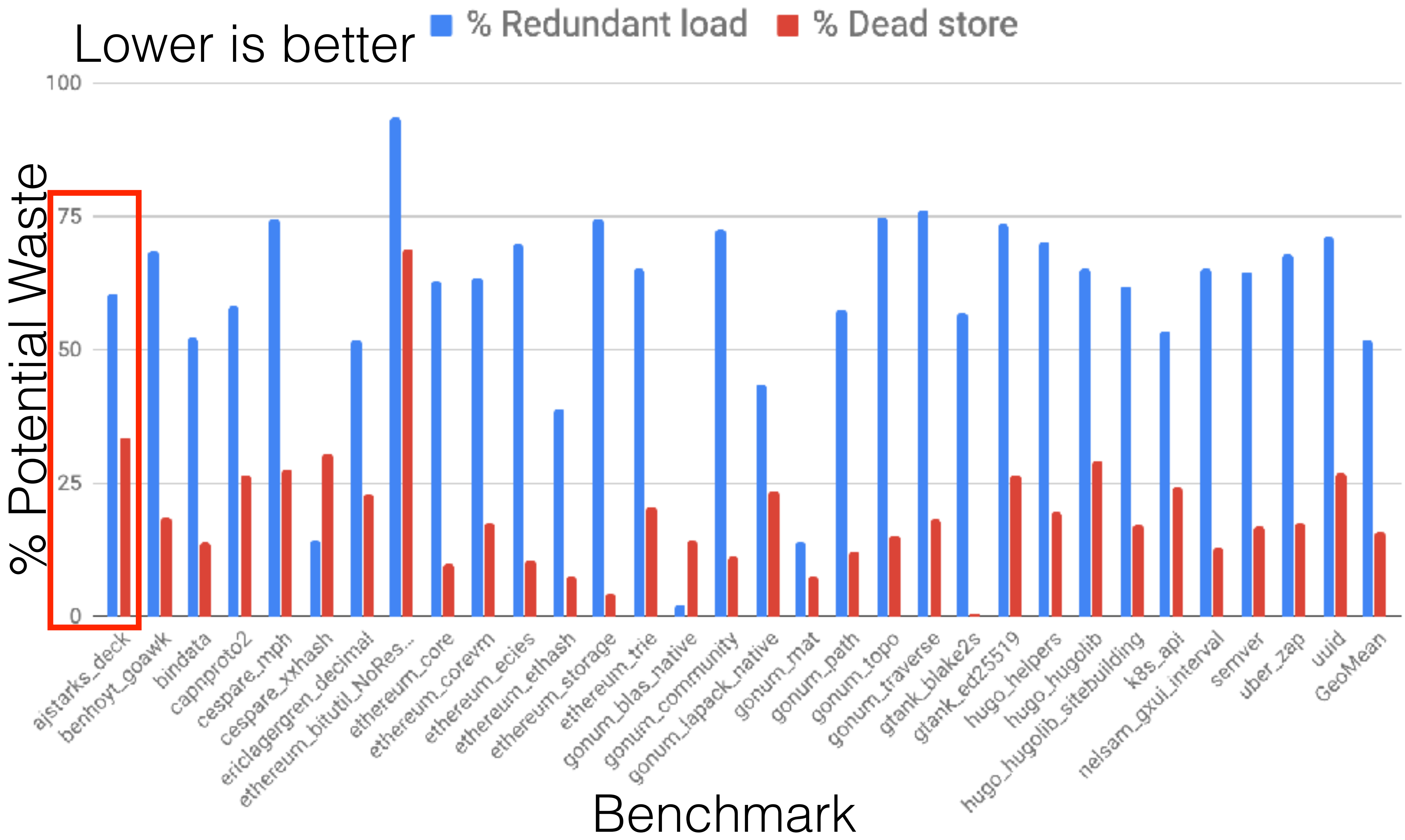
Optimization:
reuse previous via if r2 is changed
from the previous iteration

1.5x speedup

Wasteful Memory Operations in Go Benchmarks

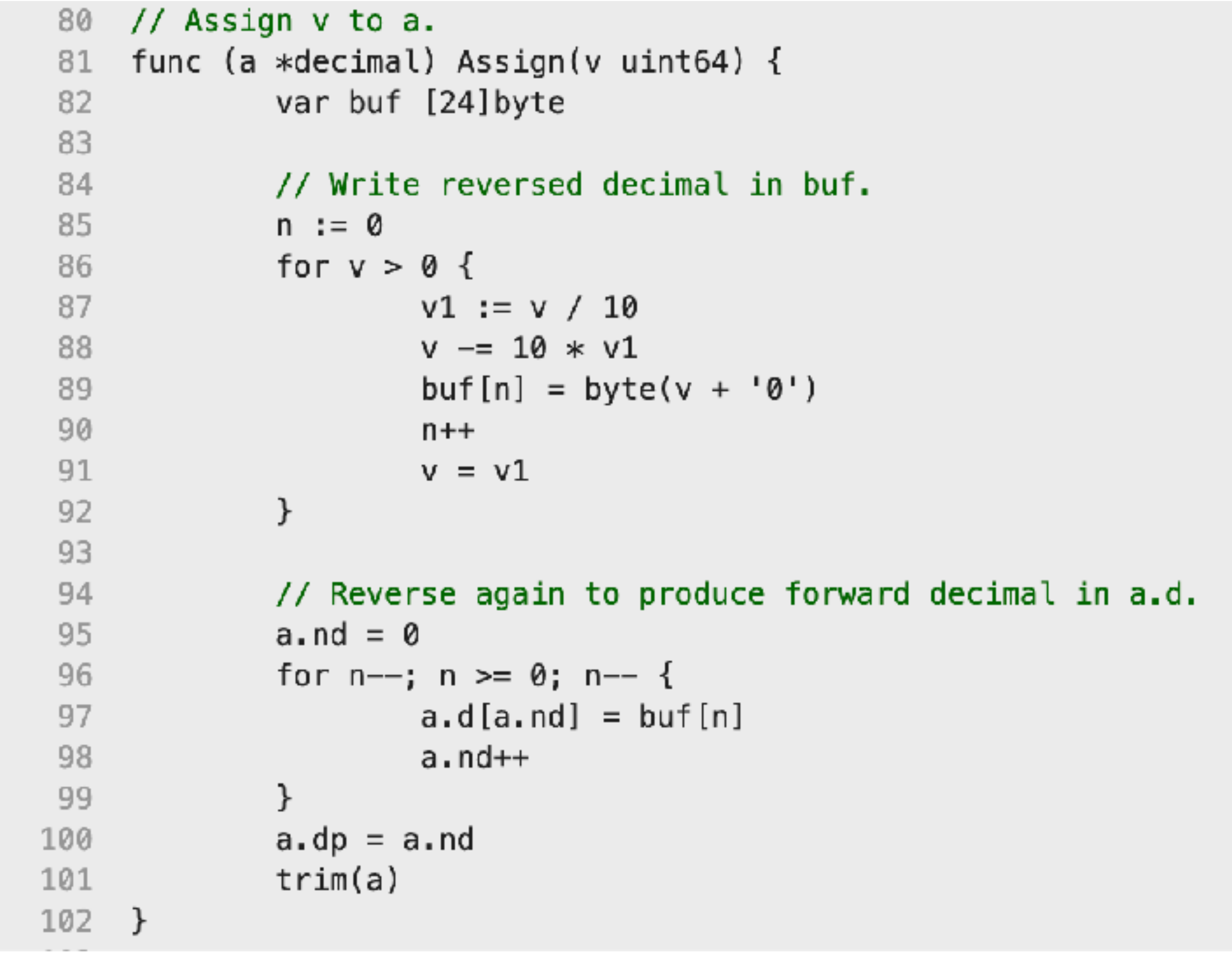

Lower is better

Wasteful Memory Operations in Go Benchmarks

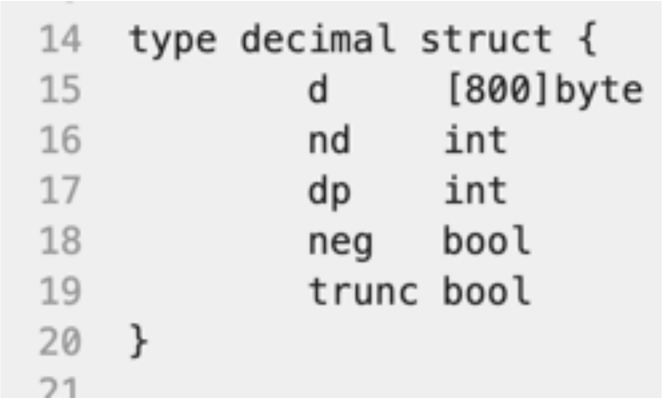


Lower is better

Inefficiencies in Go Lang strconv/decimal.go



```go
// Assign v to a.
func (a *decimal) Assign(v uint64) {
	var buf [24]byte

	// Write reversed decimal in buf.
	n := 0
	for v > 0 {
		v1 := v / 10
		v -= 10 * v1
		buf[n] = byte(v + '0')
		n++
		v = v1
	}

	// Reverse again to produce forward decimal in a.d.
	a.nd = 0
	for n--; n >= 0; n-- {
		a.d[a.nd] = buf[n]
		a.nd++
	}
	a.dp = a.nd
	trim(a)
}
```

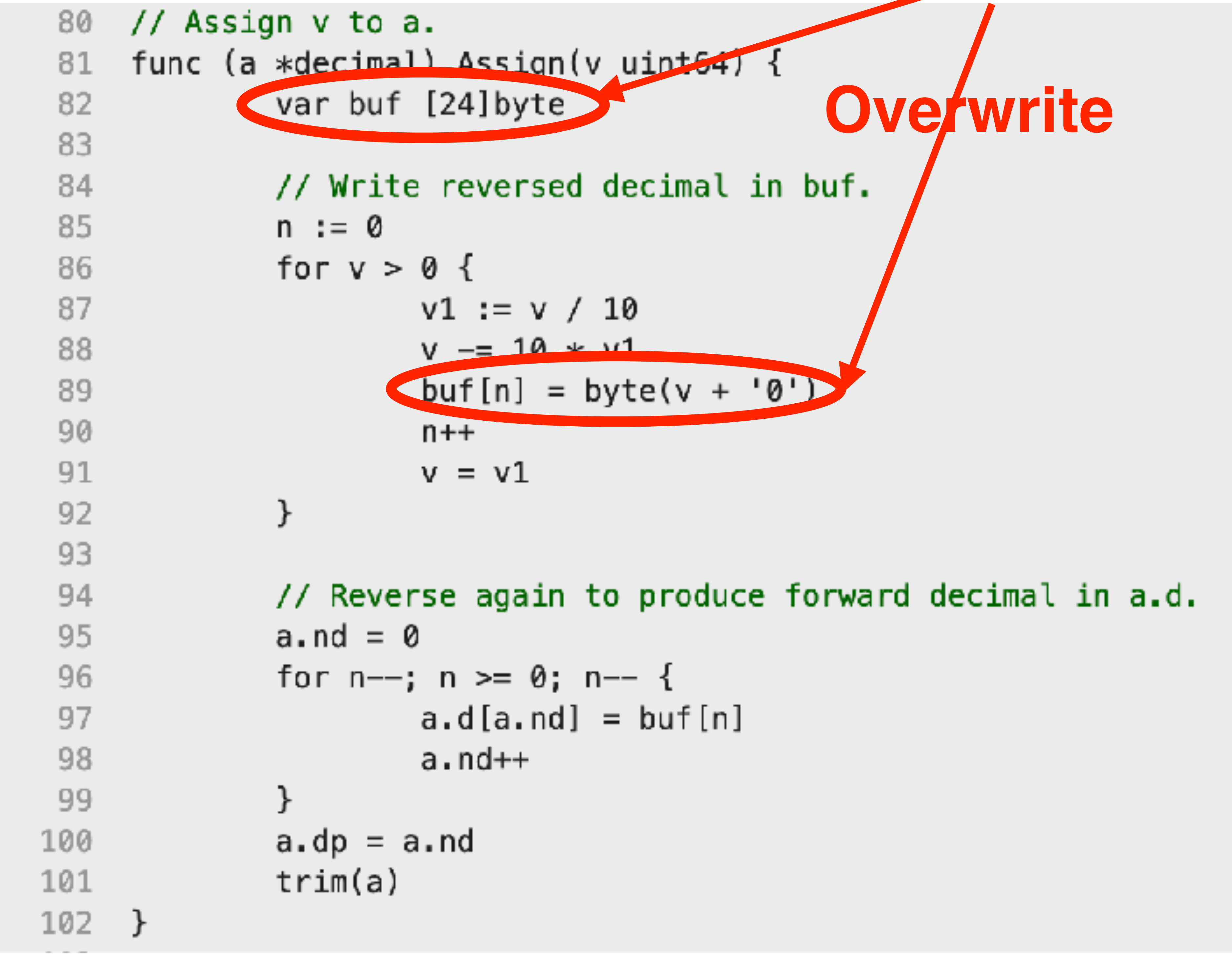


```go
type decimal struct {
	d     [800]byte
	nd    int
	dp    int
	neg   bool
	trunc bool
}
```

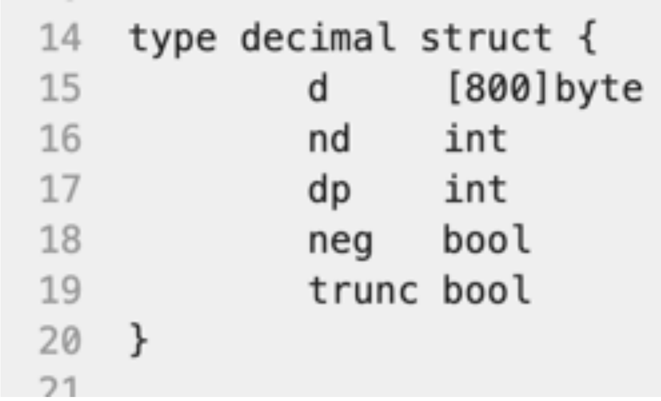
Inefficiencies in Go Lang strconv/decimal.go

Wasteful zero initialization

Overwrite

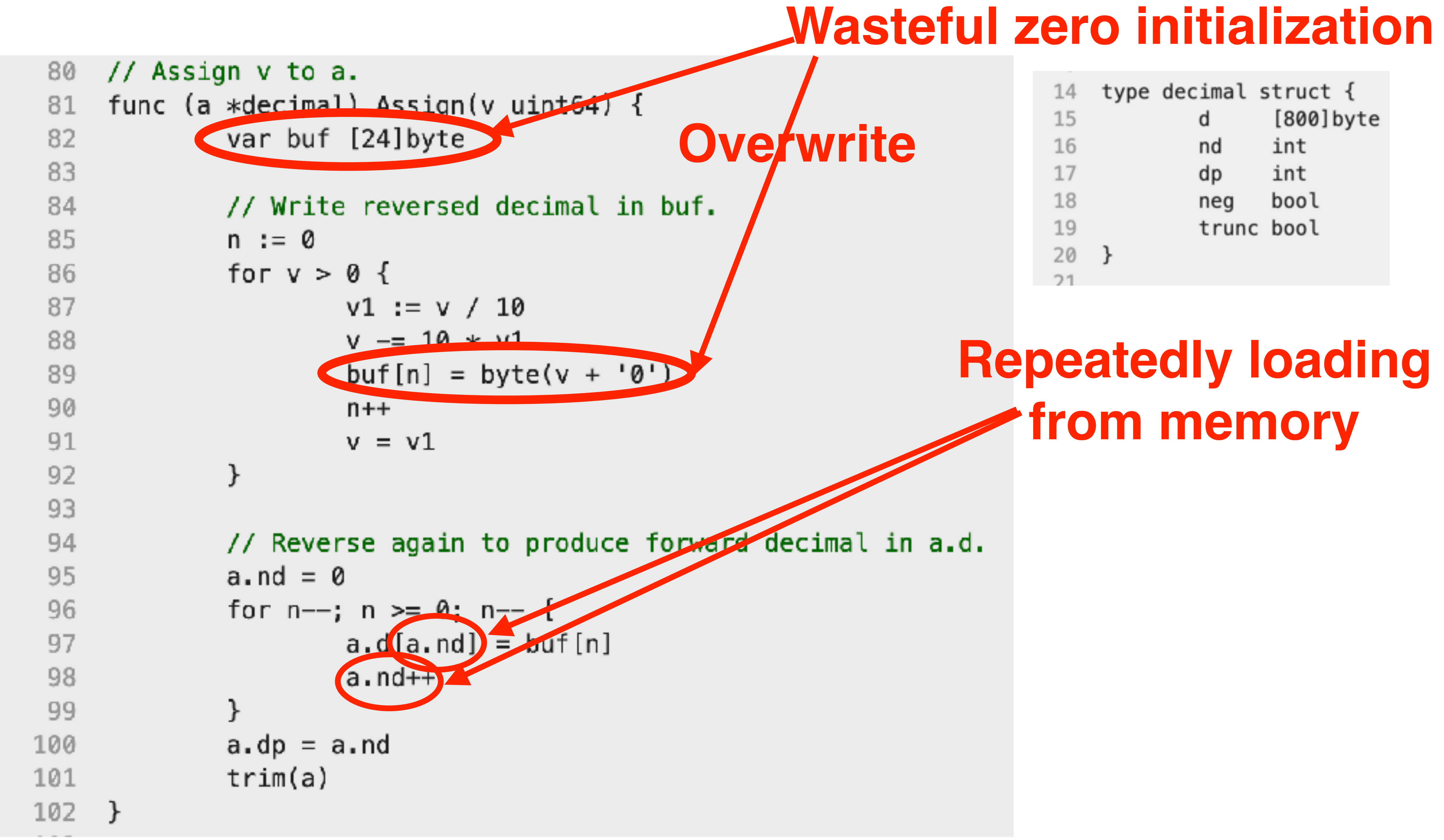


```
// Assign v to a.
func (a *decimal) Assign(v uint64) {
	var buf [24]byte

	// Write reversed decimal in buf.
	n := 0
	for v > 0 {
		v1 := v / 10
		v -= 10 * v1
		buf[n] = byte(v + '0')
		n++
		v = v1
	}

	// Reverse again to produce forward decimal in a.d.
	a.nd = 0
	for n--; n >= 0; n-- {
		a.d[a.nd] = buf[n]
		a.nd++
	}
	a.dp = a.nd
	trim(a)
}
```



```
type decimal struct {
	d     [800]byte
	nd    int
	dp    int
	neg   bool
	trunc bool
}
```

Inefficiencies in Go Lang strconv/decimal.go



```go
// Assign v to a.
func (a *decimal) Assign(v uint64) {
        var buf [24]byte

        // Write reversed decimal in buf.
        n := 0
        for v > 0 {
                v1 := v / 10
                v -= 10 * v1
                buf[n] = byte(v + '0')
                n++
                v = v1
        }

        // Reverse again to produce forward decimal in a.d.
        a.nd = 0
        for n--; n >= 0; n-- {
                a.d[a.nd] = buf[n]
                a.nd++
        }
        a.dp = a.nd
        trim(a)
}
```

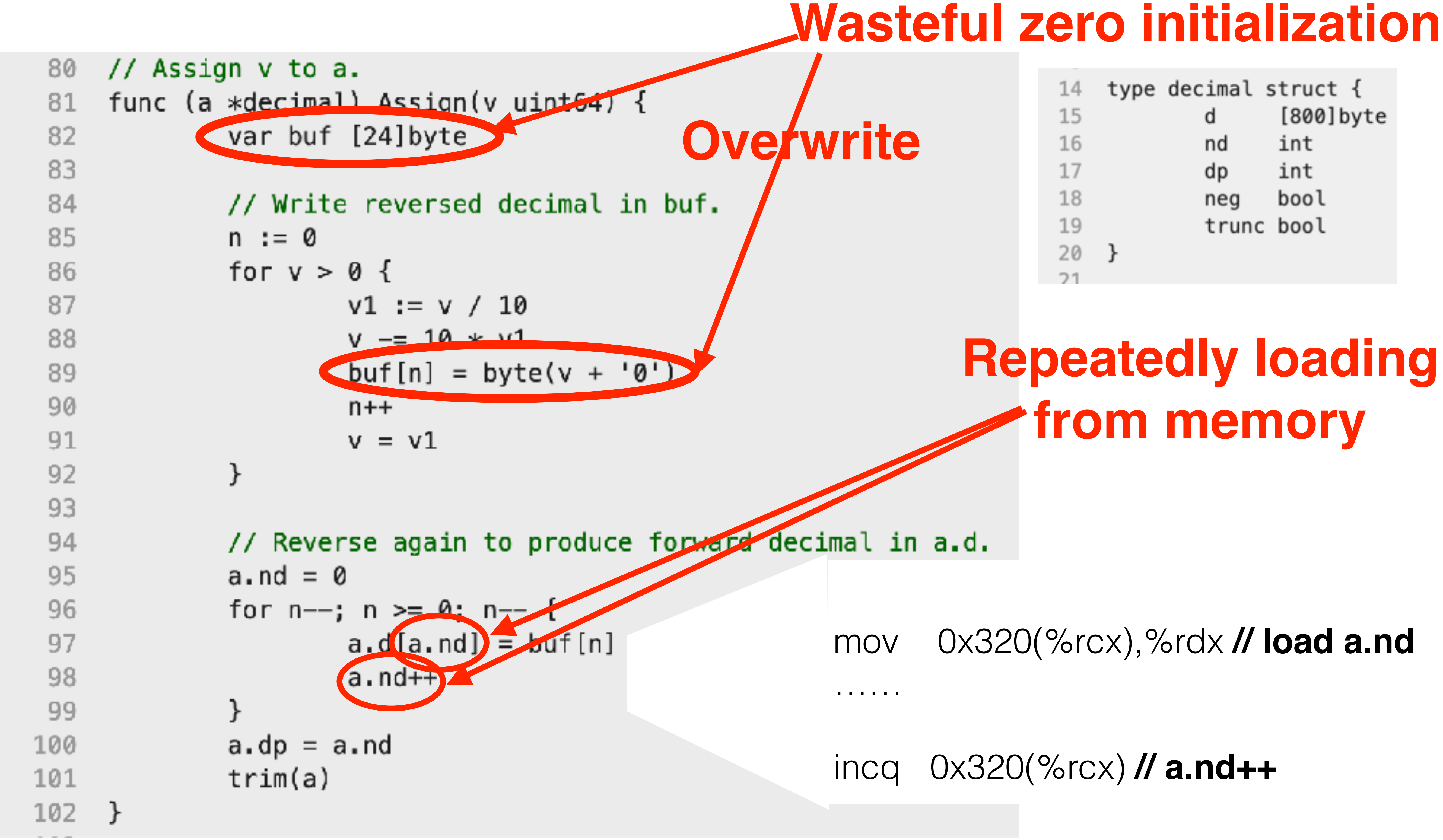
```go
type decimal struct {
        d     [800]byte
        nd    int
        dp    int
        neg   bool
        trunc bool
}
```

Wasteful zero initialization

Overwrite

Repeatedly loading from memory

Inefficiencies in Go Lang strconv/decimal.go



Wasteful zero initialization

Overwrite

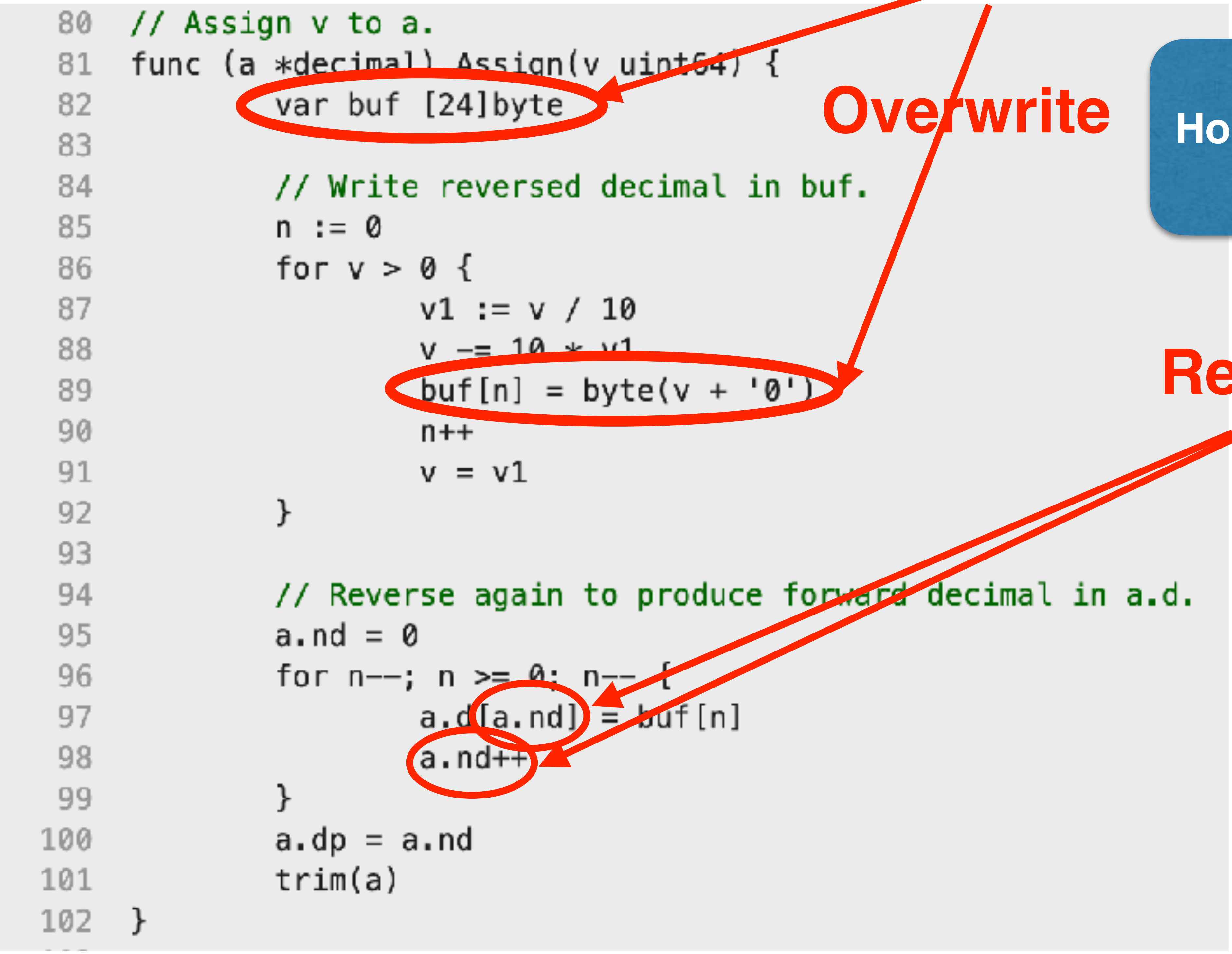
```go
// Assign v to a.
func (a *decimal) Assign(v uint64) {
	var buf [24]byte

	// Write reversed decimal in buf.
	n := 0
	for v > 0 {
		v1 := v / 10
		v -= 10 * v1
		buf[n] = byte(v + '0')
		n++
		v = v1
	}

	// Reverse again to produce forward decimal in a.d.
	a.nd = 0
	for n--; n >= 0; n-- {
		a.d[a.nd] = buf[n]
		a.nd++
	}
	a.dp = a.nd
	trim(a)
}
```

```go
type decimal struct {
	d     [800]byte
	nd    int
	dp    int
	neg   bool
	trunc bool
}
```

Repeatedly loading from memory

```
mov    0x320(%rcx),%rdx // load a.nd
......

incq   0x320(%rcx) // a.nd++
```

Inefficiencies in Go Lang strconv/decimal.go



```
// Assign v to a.
func (a *decimal) Assign(v uint64) {
        var buf [24]byte

        // Write reversed decimal in buf.
        n := 0
        for v > 0 {
                v1 := v / 10
                v -= 10 * v1
                buf[n] = byte(v + '0')
                n++
                v = v1
        }

        // Reverse again to produce forward decimal in a.d.
        a.nd = 0
        for n--; n >= 0; n-- {
                a.d[a.nd] = buf[n]
                a.nd++
        }
        a.dp = a.nd
        trim(a)
}
```

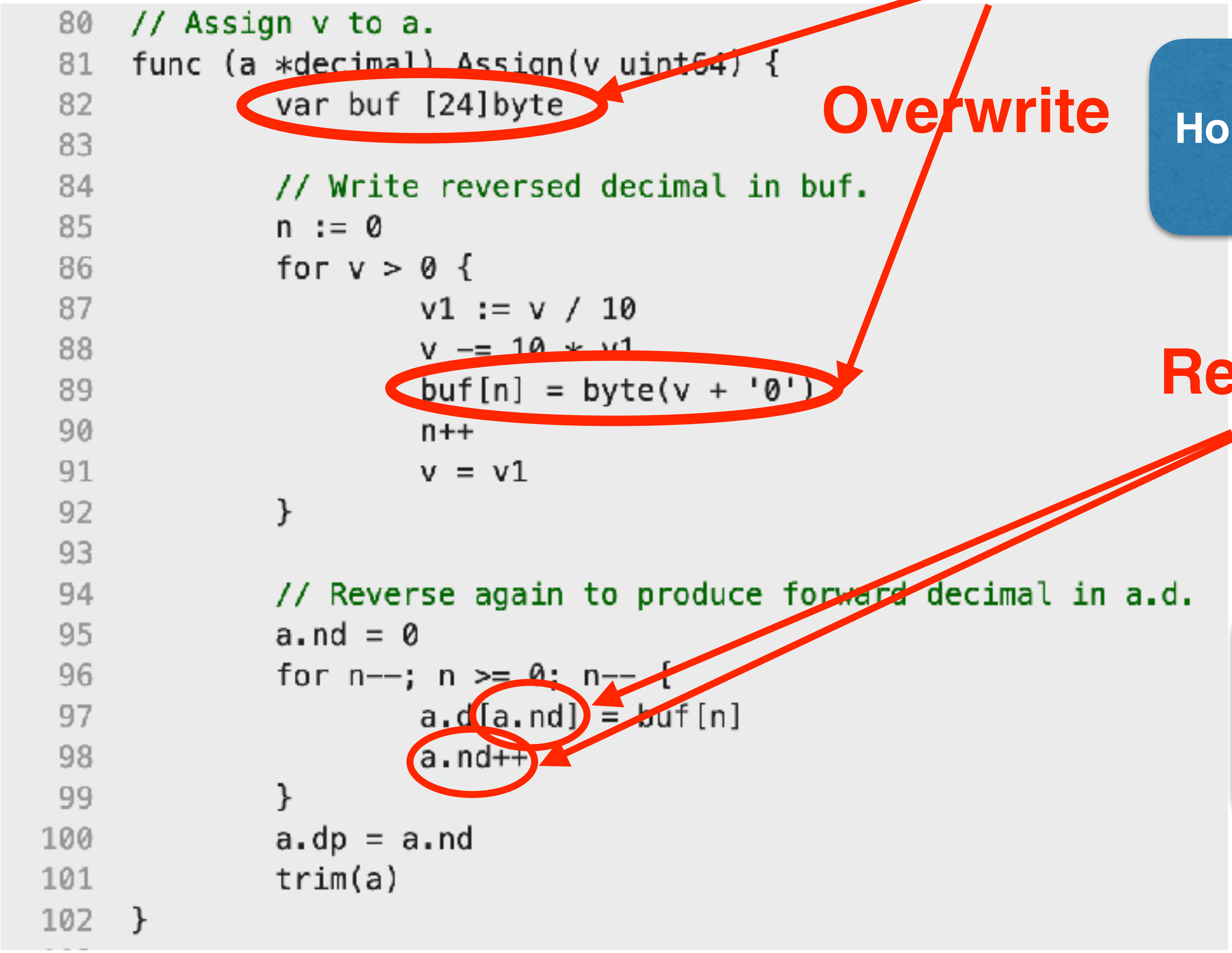
Wasteful zero initialization

Overwrite

Repeatedly loading from memory

(thread unsafe)
Hoist buffer to module level
⇒ 25 % speedup

Inefficiencies in Go Lang strconv/decimal.go



```
// Assign v to a.
func (a *decimal) Assign(v uint64) {
        var buf [24]byte

        // Write reversed decimal in buf.
        n := 0
        for v > 0 {
                v1 := v / 10
                v -= 10 * v1
                buf[n] = byte(v + '0')
                n++
                v = v1
        }

        // Reverse again to produce forward decimal in a.d.
        a.nd = 0
        for n--; n >= 0; n-- {
                a.d[a.nd] = buf[n]
                a.nd++
        }
        a.dp = a.nd
        trim(a)
}
```

Wasteful zero initialization

Overwrite

Repeatedly loading from memory

(thread unsafe)
Hoist buffer to module level
⇒ 25 % speedup

Use CPU registers
⇒ 40% speedup