Automated Debugging for Arbitrarily Long Executions

Cristian Zamfir, Baris Kasikci, Johannes Kinder, Edouard Bugnion, George Candea

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Debugging is Hard

- Debugging = diagnose + fix the root cause
- May take days-months to diagnose bugs in the real world[1]

[1] **Concurrency at Microsoft – An Exploratory Survey**, *CAV workshop* 2008

Real World Debugging

Debugging during development

Real World Debugging

Debugging during development

```
$ gdb ./program
```

Real World Debugging

Debugging during development

```
$ gdb ./program
(gdb) record
```

Real World Debugging

Debugging during development

```
$ gdb ./program
(gdb) record
(gdb) run
```

Real World Debugging

Debugging during development

```
$ gdb ./program
(gdb) record
(gdb) run
Segmentation fault
```

Real World Debugging

Debugging during development

```
$ gdb ./program
(gdb) record
(gdb) run
Segmentation fault
(gdb) reverse-step
```

Real World Debugging

Debugging during development

```
$ gdb ./program
(gdb) record
(gdb) run
Segmentation fault
(gdb) reverse-step
```

Real World Debugging

Debugging during development

Debugging in the real world

```
$ gdb ./program
(gdb) record
(gdb) run
Segmentation fault
(gdb) reverse-step
```

Real World Debugging

Debugging during development

```
$ gdb ./program
(gdb) record
(gdb) run
Segmentation fault
(gdb) reverse-step
```

Debugging in the real world

```
$ ./program
```

Real World Debugging

Debugging during development

```
$ gdb ./program
(gdb) record
(gdb) run
Segmentation fault
(gdb) reverse-step
```

Debugging in the real world

```
$ ./program
Segmentation fault
```

Real World Debugging

Debugging during development

```
$ gdb ./program
(gdb) record
(gdb) run
Segmentation fault
(gdb) reverse-step
```

Debugging in the real world

```
$ ./program
Segmentation fault
(core dumped)
```

```
$ gdb ./program core
```

Real World Debugging

Debugging during development

```
$ gdb ./program
(gdb) record
(gdb) run
Segmentation fault
(gdb) reverse-step
```

Debugging in the real world

```
$ ./program
Segmentation fault
(core dumped)
```

```
$ gdb ./program core
(gdb) reverse-step
```

Real World Debugging

Debugging during development

```
$ gdb ./program
(gdb) record
(gdb) run
Segmentation fault
(gdb) reverse-step
```

Debugging in the real world

```
$ ./program
Segmentation fault
(core dumped)
```

```
$ gdb ./program core
(gdb) reverse-step
Target core command unsupported
```

Debug Without Recording

Debug Without Recording

What are the classes of information necessary for debugging?

Debug Without Recording

Coredump

10101010 10101011

+

Program

11101011 10001001

What are the classes of information necessary for debugging?

Debug Without Recording

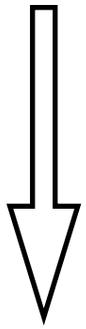
Coredump

10101010
10101011

+

Program

11101011
10001001



Synthesize

program inputs

thread schedule

What are the classes of information necessary for debugging?

Debug Without Recording

Coredump

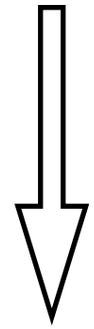
10101010
10101011

+

Program

11101011
10001001

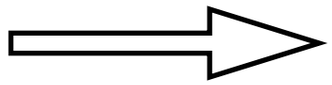
What are the classes of information necessary for debugging?

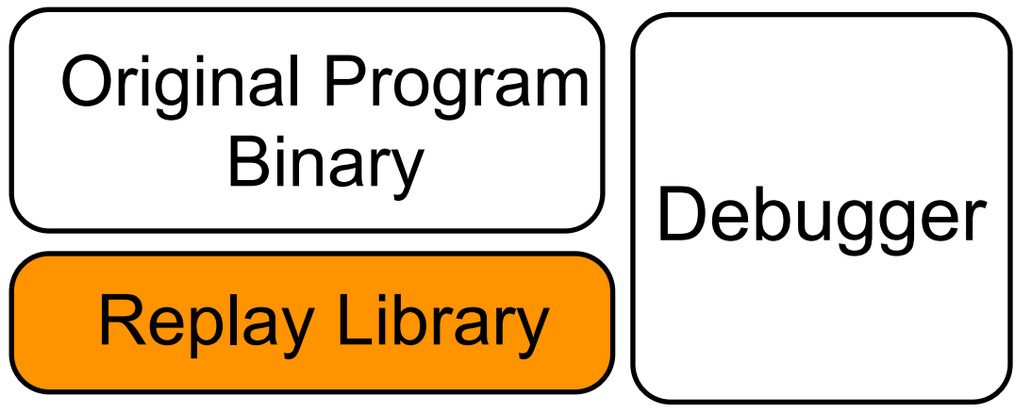


Synthesize

program inputs

thread schedule





Debug Without Recording

Synthesize

program inputs

thread schedule

Debug Without Recording

Synthesize

program inputs

thread schedule

Debug Without Recording

Synthesize

program inputs

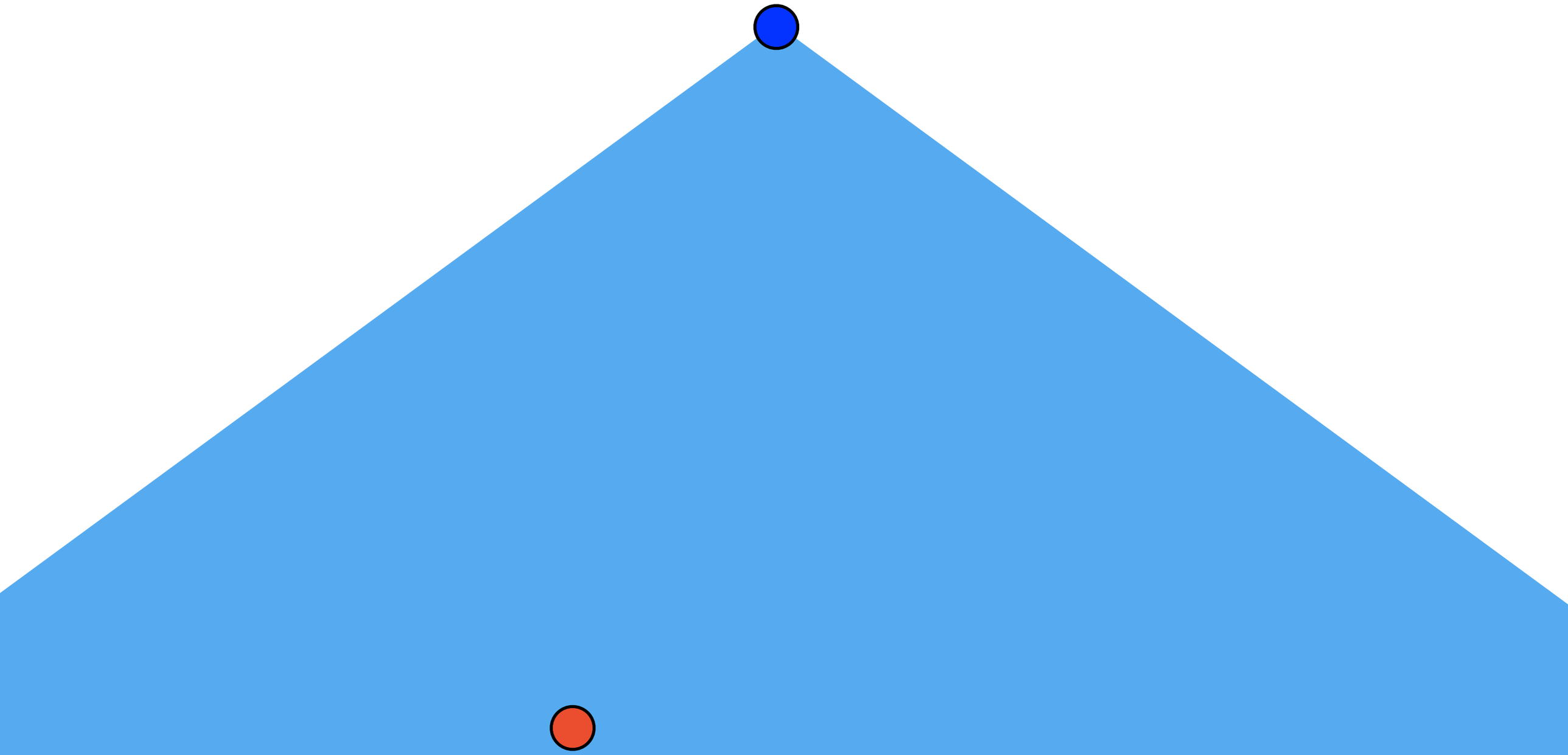
thread schedule

Debug Without Recording

Synthesize

program inputs

thread schedule



Debug Without Recording

Synthesize

program inputs

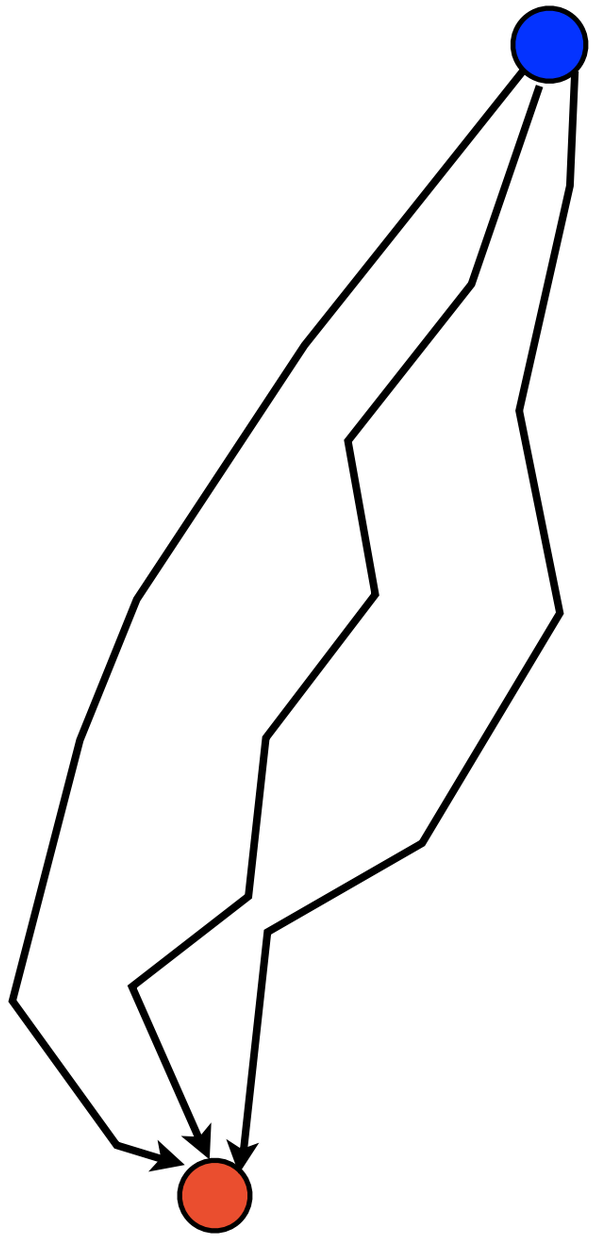
thread schedule

Debug Without Recording

Synthesize

program inputs

thread schedule

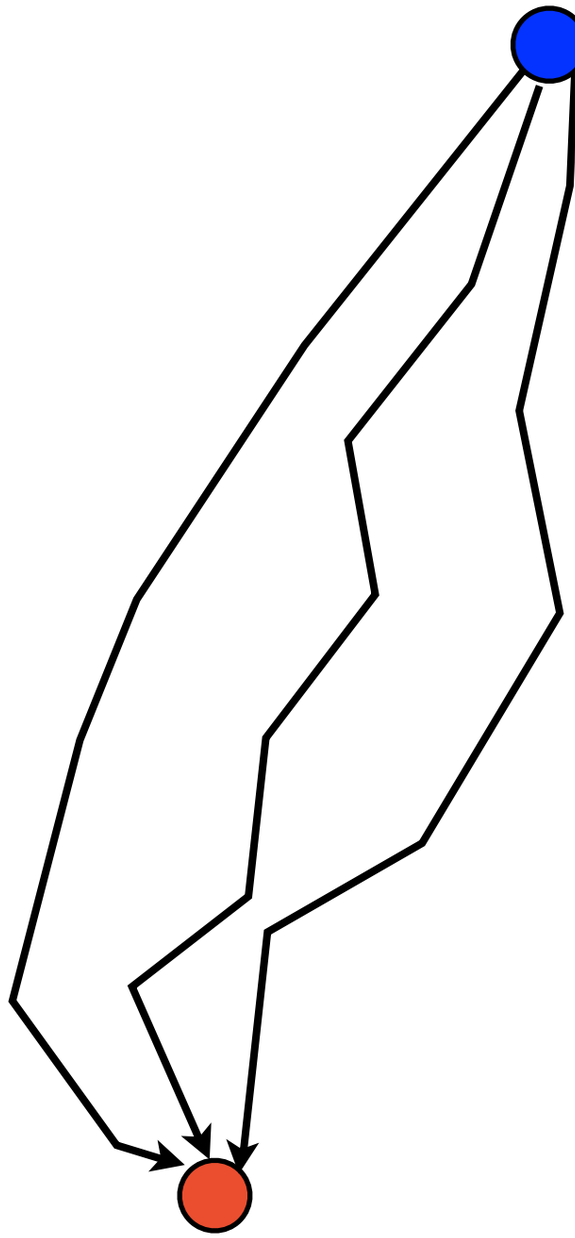


Debug Without Recording

Synthesize

program inputs

thread schedule



Exact same execution is not necessary

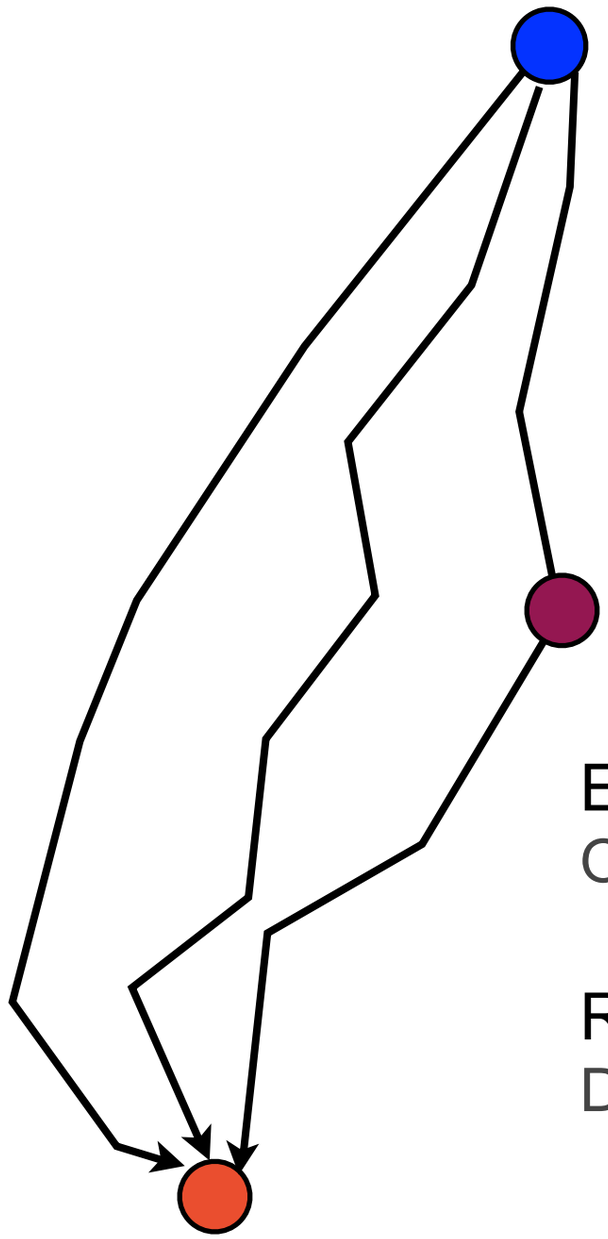
ODR, PRES (SOSP'09) and ESD (EuroSys'10)

Debug Without Recording

Synthesize

program inputs

thread schedule



Exact same execution is not necessary
ODR, PRES (SOSP'09) and ESD (EuroSys'10)

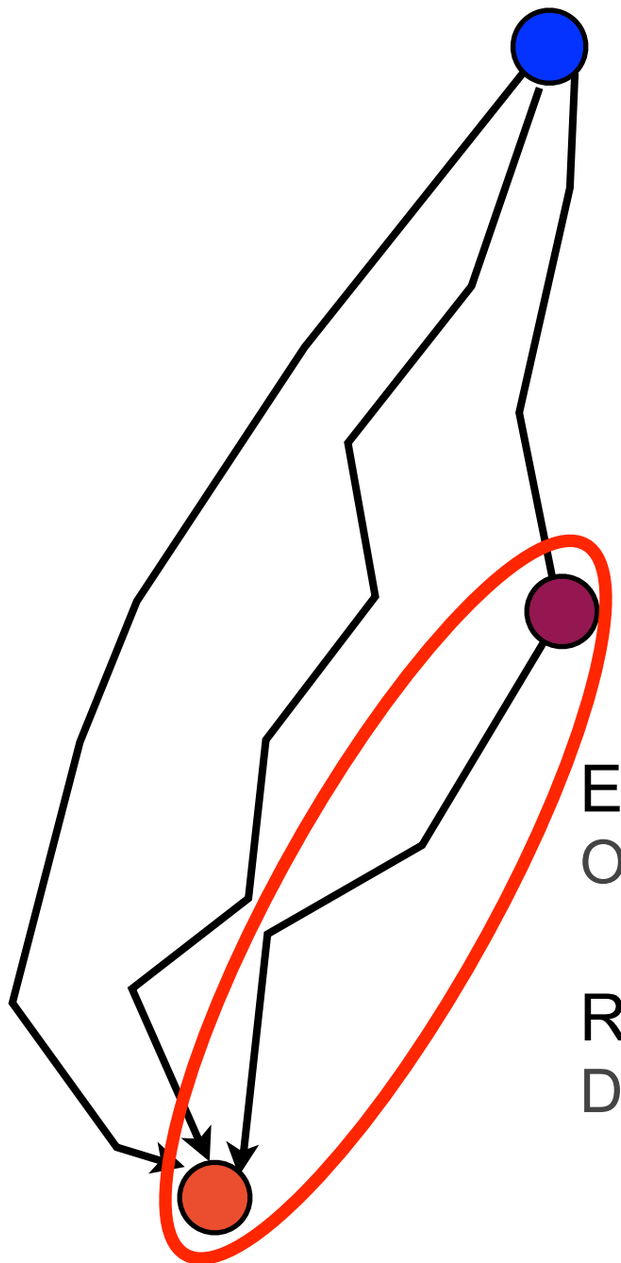
Reproduce the root cause and the failure
Debug Determinism (HotOS'11)

Debug Without Recording

Synthesize

program inputs

thread schedule



Exact same execution is not necessary

ODR, PRES (SOSP'09) and ESD (EuroSys'10)

Reproduce the root cause and the failure

Debug Determinism (HotOS'11)

Reverse Execution Synthesis

Synthesize

program inputs

thread schedule

Reverse Execution Synthesis

Synthesize

program inputs

thread schedule

The root cause is close to the failure

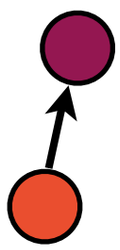
85% of the time (Conseq, ASPLOS'11)

Reverse Execution Synthesis

Synthesize

program inputs

thread schedule



The root cause is close to the failure

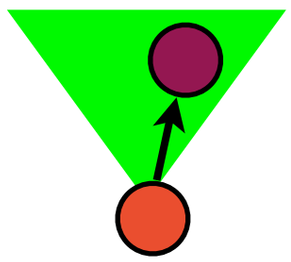
85% of the time (Conseq, ASPLOS'11)

Reverse Execution Synthesis

Synthesize

program inputs

thread schedule



The root cause is close to the failure

85% of the time (Conseq, ASPLOS'11)

Reverse Execution Synthesis

Synthesize

program inputs

thread schedule

The root cause is close to the failure

85% of the time (Conseq, ASPLOS'11)

Reverse Execution Synthesis

Reverse Execution Synthesis

```
x = 1;
if (f(x) == y) {
  goto next;
}
...
x = 2;
if (g(x) == y) {
  goto next;
}
exit();
next:
  buffer[y] = 1
```

Reverse Execution Synthesis

```
x = 1;
if (f(x) == y) {
  goto next;
}
...
x = 2;
if (g(x) == y) {
  goto next;
}
exit();
next:
  buffer[y] = 1
```

Coredump:

x	1
y	10

Reverse Execution Synthesis

```
x = 1;
if (f(x) == y) {
  goto next;
}
...
x = 2;
if (g(x) == y) {
  goto next;
}
exit();
next:
  buffer[y] = 1
```

```
buffer[y] = 1;
```

Coredump: (buffer overflow)

x	1
y	10

Reverse Execution Synthesis

```
x = 1;
if (f(x) == y) {
  goto next;
}
...
x = 2;
if (g(x) == y) {
  goto next;
}
exit();
next:
  buffer[y] = 1
```

```
x = 1;
if (f(x) == y)
```

True

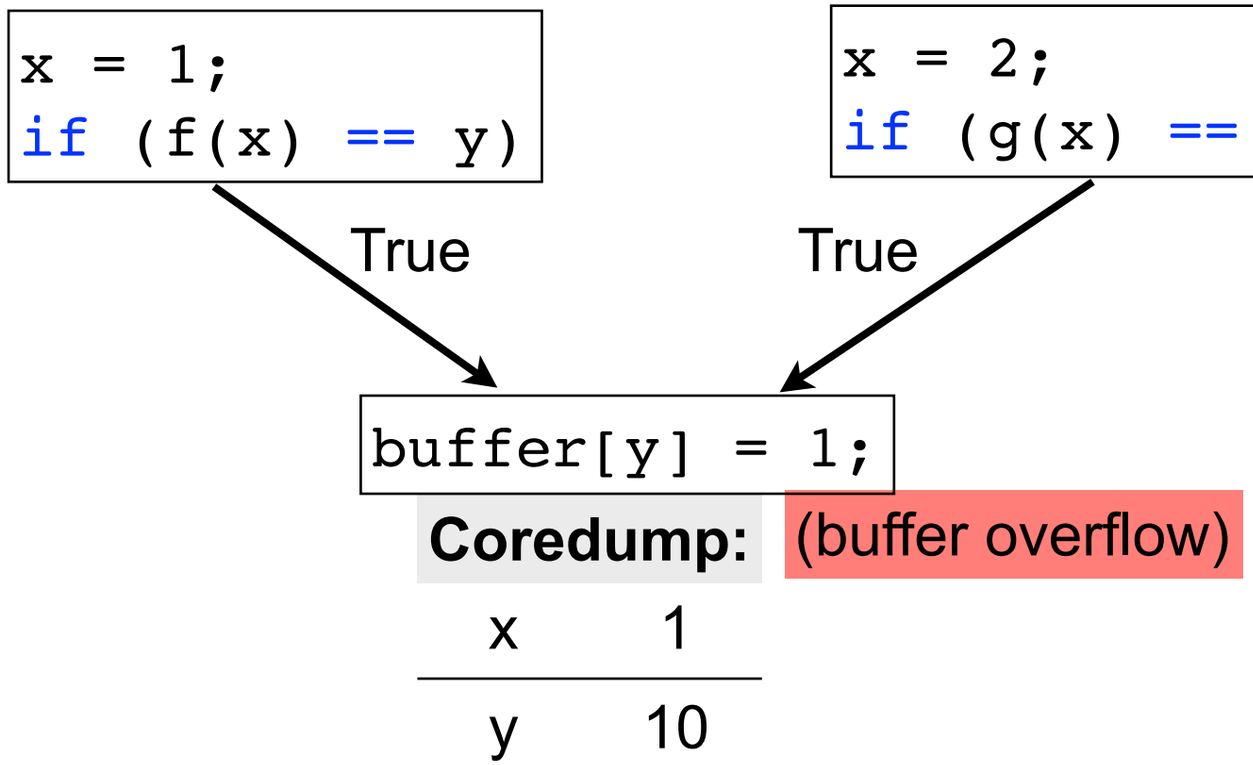
```
buffer[y] = 1;
```

Coredump: (buffer overflow)

x	1
y	10

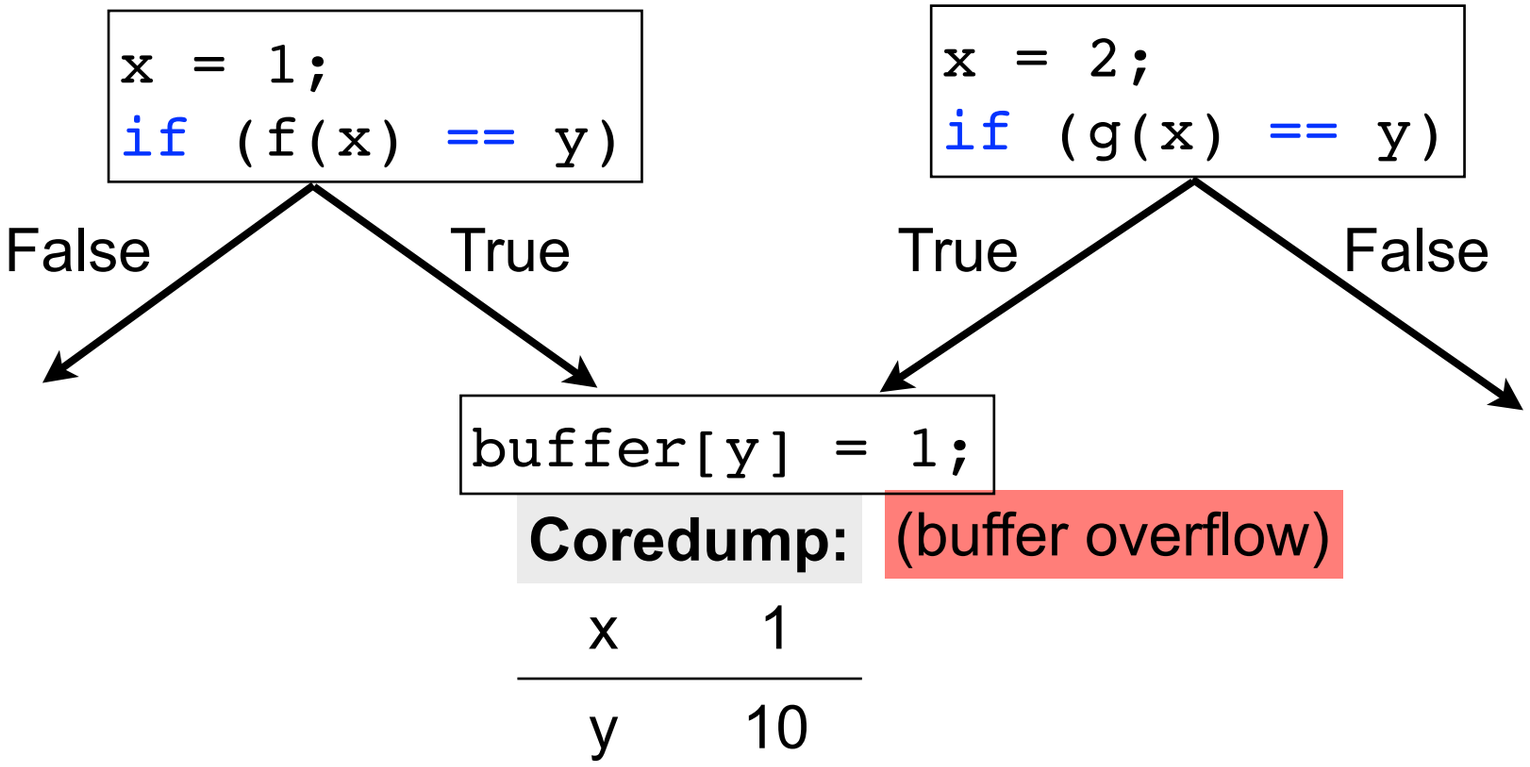
Reverse Execution Synthesis

```
x = 1;
if (f(x) == y) {
  goto next;
}
...
x = 2;
if (g(x) == y) {
  goto next;
}
exit();
next:
  buffer[y] = 1
```



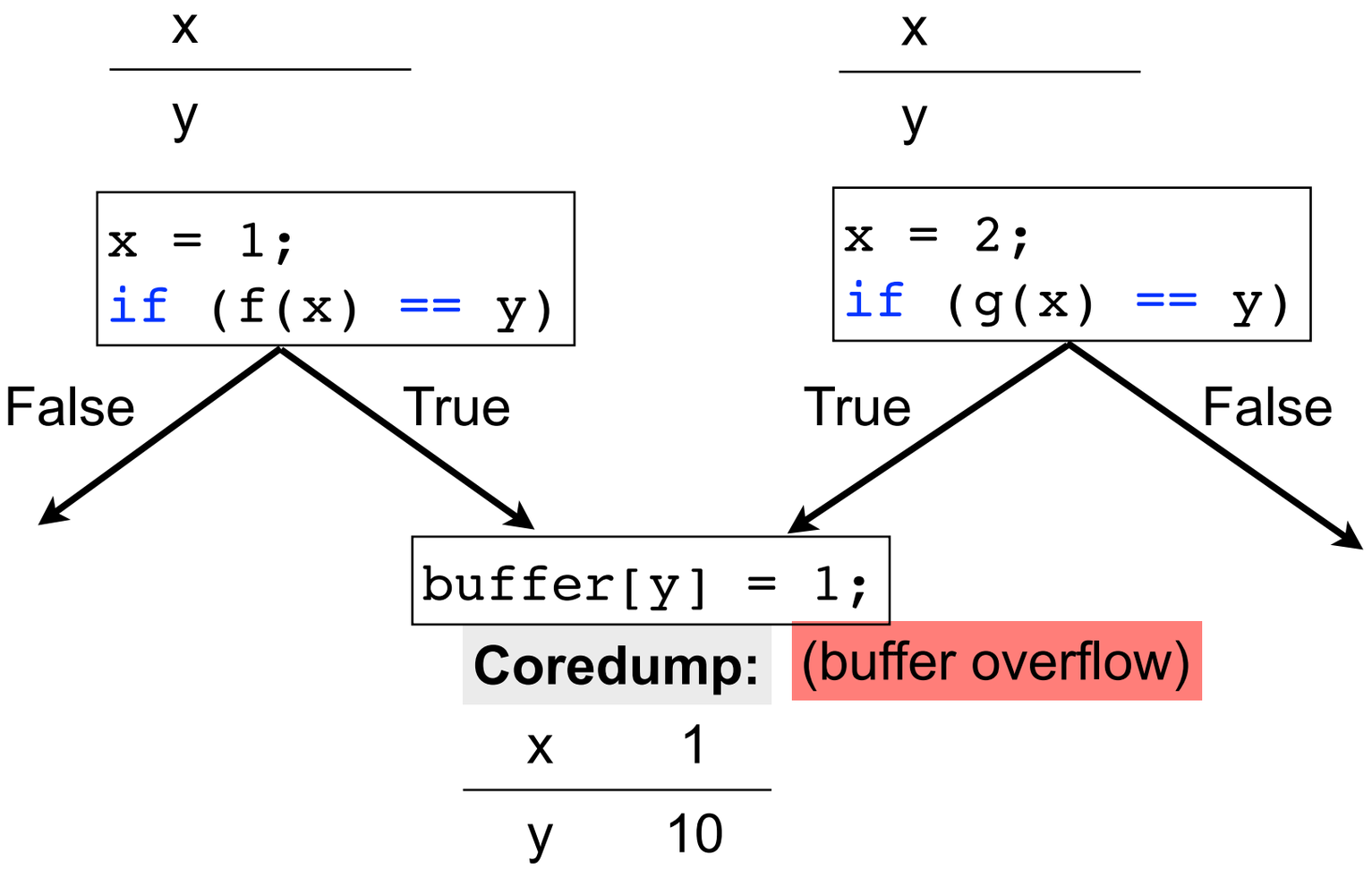
Reverse Execution Synthesis

```
x = 1;
if (f(x) == y) {
  goto next;
}
...
x = 2;
if (g(x) == y) {
  goto next;
}
exit();
next:
  buffer[y] = 1
```



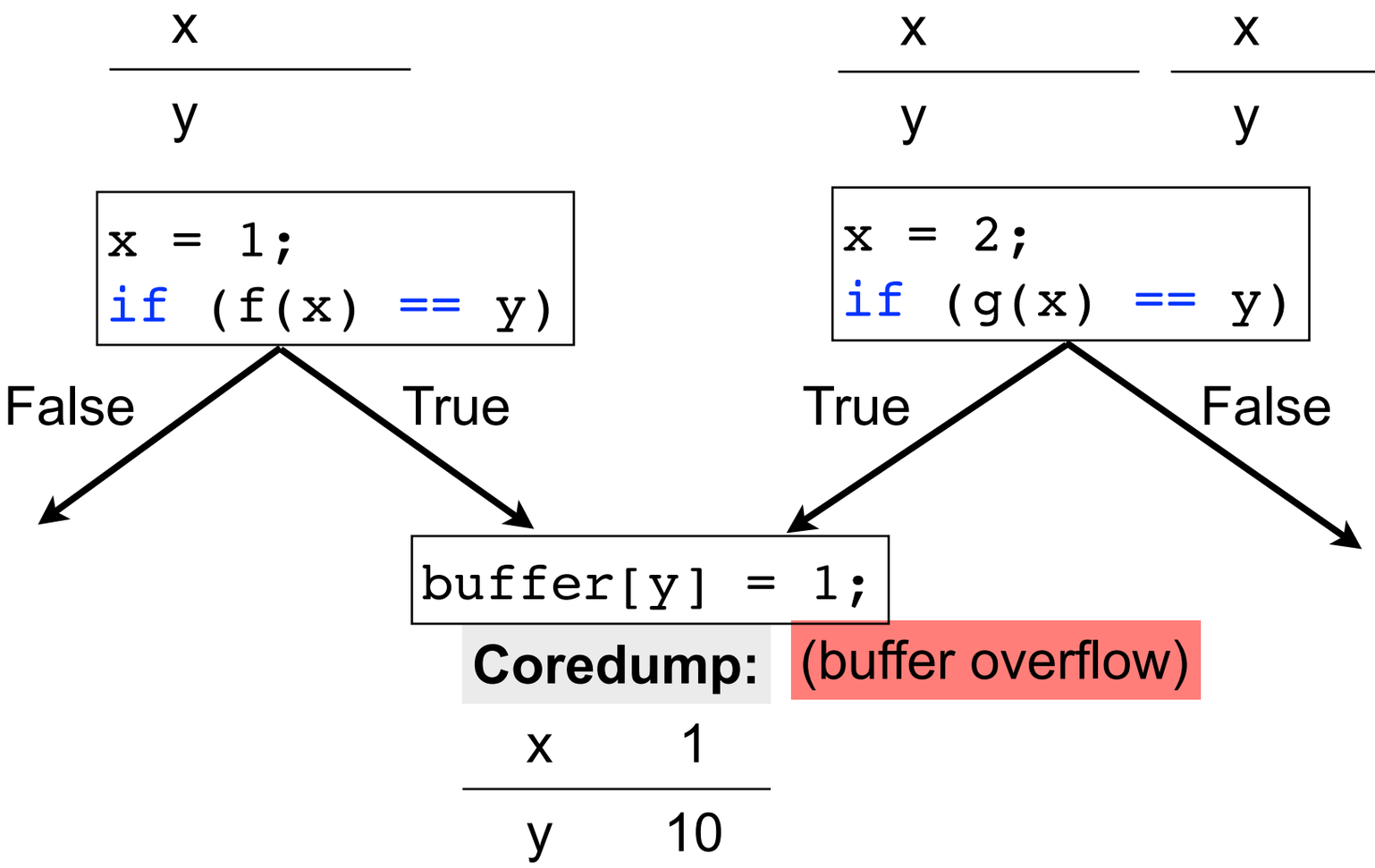
Reverse Execution Synthesis

```
x = 1;
if (f(x) == y) {
  goto next;
}
...
x = 2;
if (g(x) == y) {
  goto next;
}
exit();
next:
  buffer[y] = 1
```



Reverse Execution Synthesis

```
x = 1;
if (f(x) == y) {
  goto next;
}
...
x = 2;
if (g(x) == y) {
  goto next;
}
exit();
next:
  buffer[y] = 1
```



Reverse Execution Synthesis

```
x = 1;
if (f(x) == y) {
  goto next;
}
...
x = 2;
if (g(x) == y) {
  goto next;
}
exit();
next:
buffer[y] = 1
```

x / y

```
x = 1;
if (f(x) == y)
```

False → `buffer[y] = 1;`

True → `x = 2;`

x / y

```
x = 2;
if (g(x) == y)
```

True → `buffer[y] = 1;`

False → `exit();`

Coredump: (buffer overflow)

x	1
y	10

Reverse Execution Synthesis

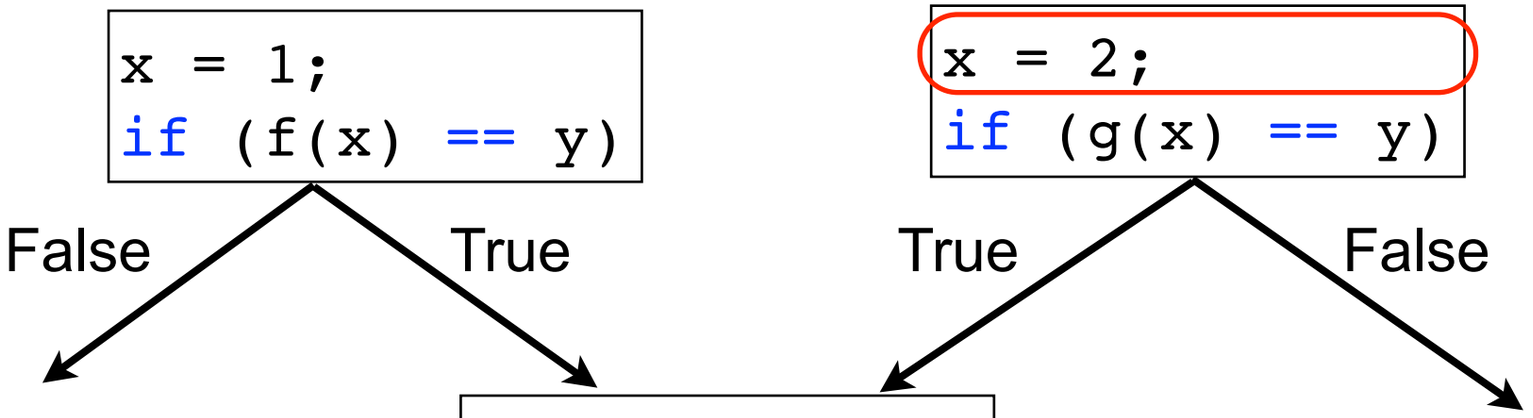
```
x = 1;
if (f(x) == y) {
  goto next;
}
...
x = 2;
if (g(x) == y) {
  goto next;
}
exit();
next:
buffer[y] = 1
```

x = 2

y = 1

x = 1

y = 10



Coredump: (buffer overflow)

Reverse Execution Synthesis

```
x = 1;
if (f(x) == y) {
  goto next;
}
...
x = 2;
if (g(x) == y) {
  goto next;
}
exit();
next:
  buffer[y] = 1
```

x	2
y	10

```
x = 1;
if (f(x) == y)
```

x	
y	

False

True

```
x = 2;
if (g(x) == y)
```

x	
y	

True

False

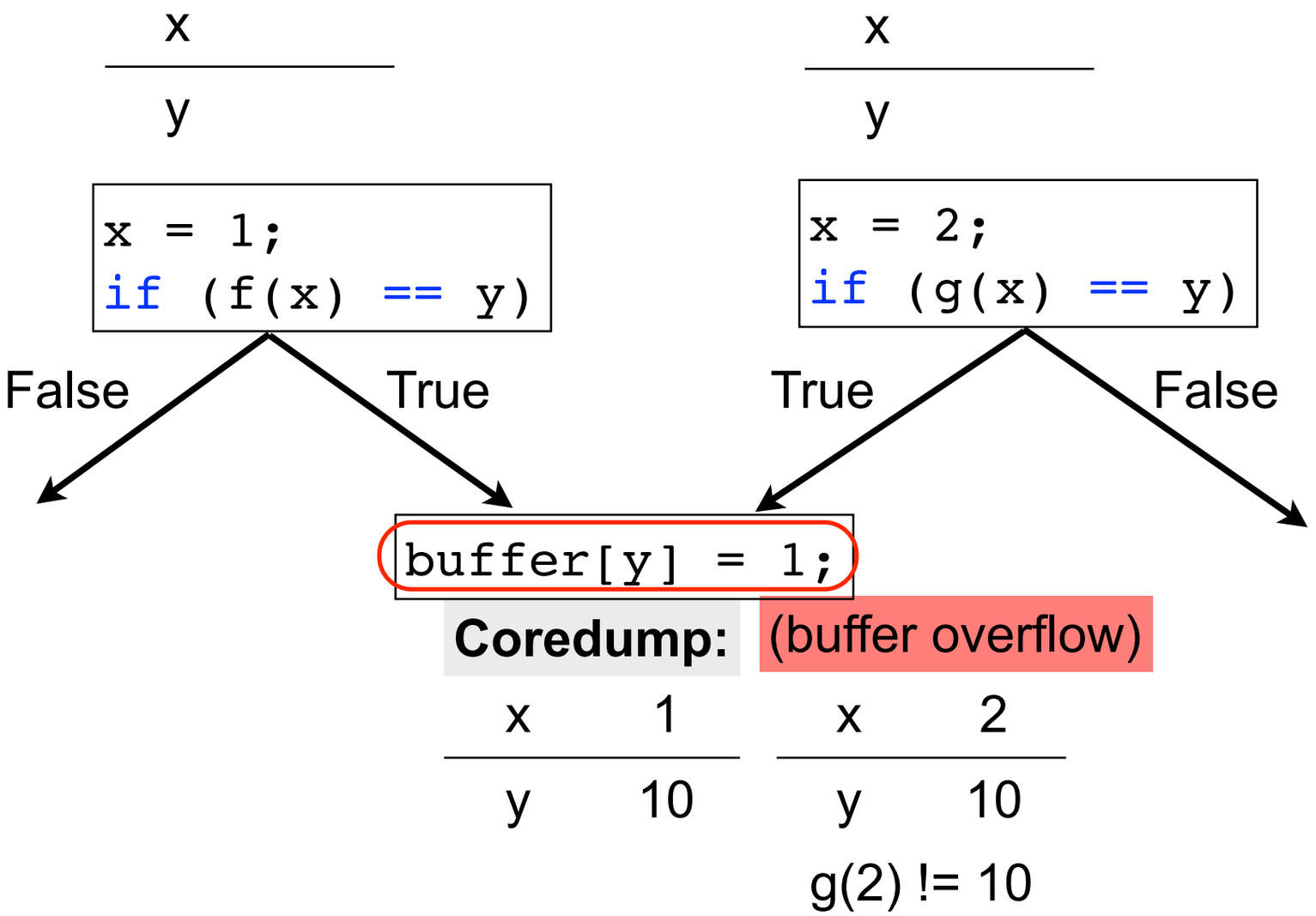
```
buffer[y] = 1;
```

Coredump: (buffer overflow)

x	1
y	10

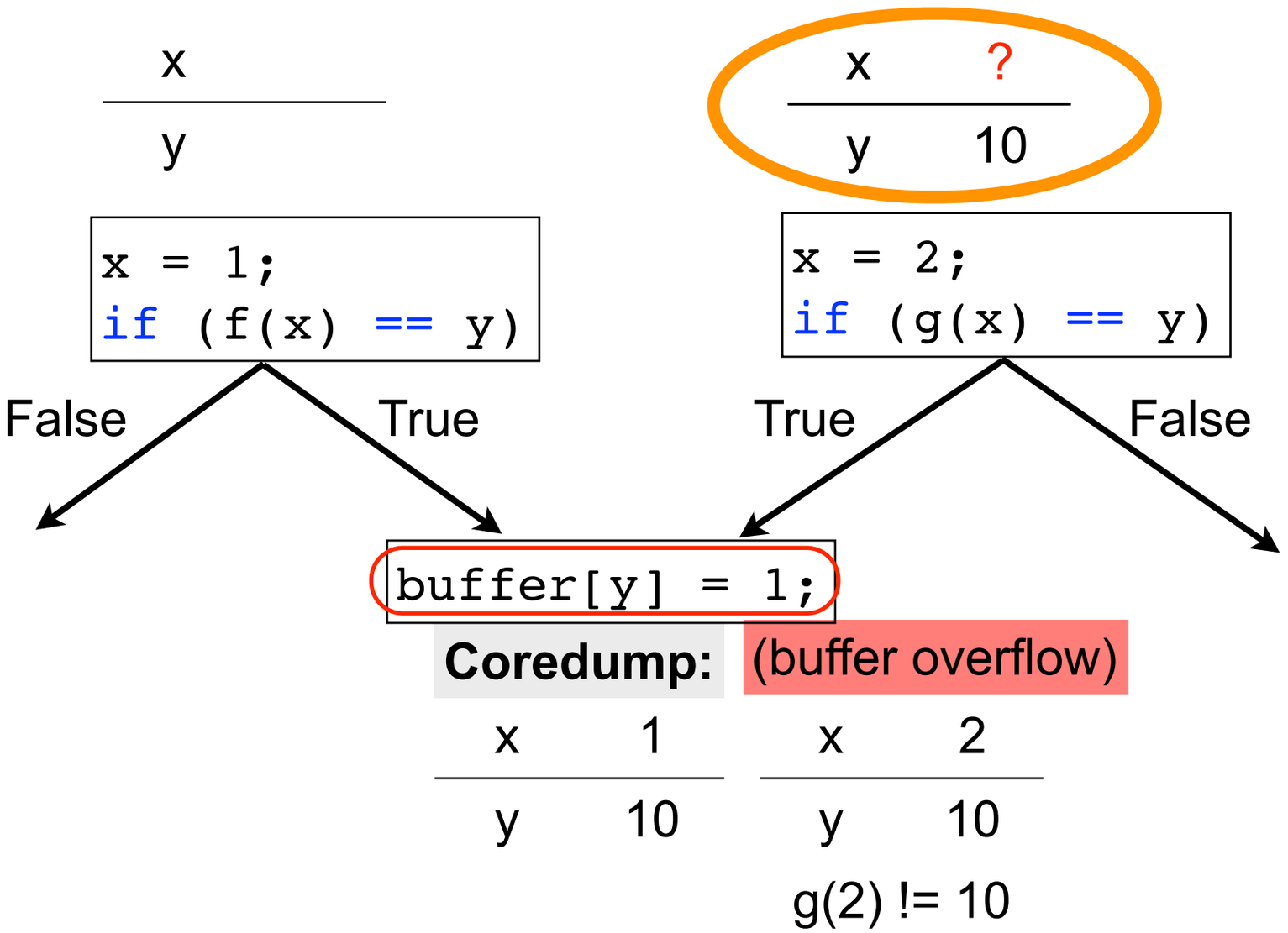
Reverse Execution Synthesis

```
x = 1;
if (f(x) == y) {
  goto next;
}
...
x = 2;
if (g(x) == y) {
  goto next;
}
exit();
next:
  buffer[y] = 1
```



Reverse Execution Synthesis

```
x = 1;
if (f(x) == y) {
  goto next;
}
...
x = 2;
if (g(x) == y) {
  goto next;
}
exit();
next:
  buffer[y] = 1
```



Reverse Execution Synthesis

```
x = 1;
if (f(x) == y) {
  goto next;
}
...
x = 2;
if (g(x) == y) {
  goto next;
}
exit();
next:
  buffer[y] = 1
```

x
y

```
x = 1;
if (f(x) == y)
```

False

True

```
buffer[y] = 1;
```

x	?
y	10

```
x = 2;
if (g(x) == y)
```

True

False

Coredump: (buffer overflow)

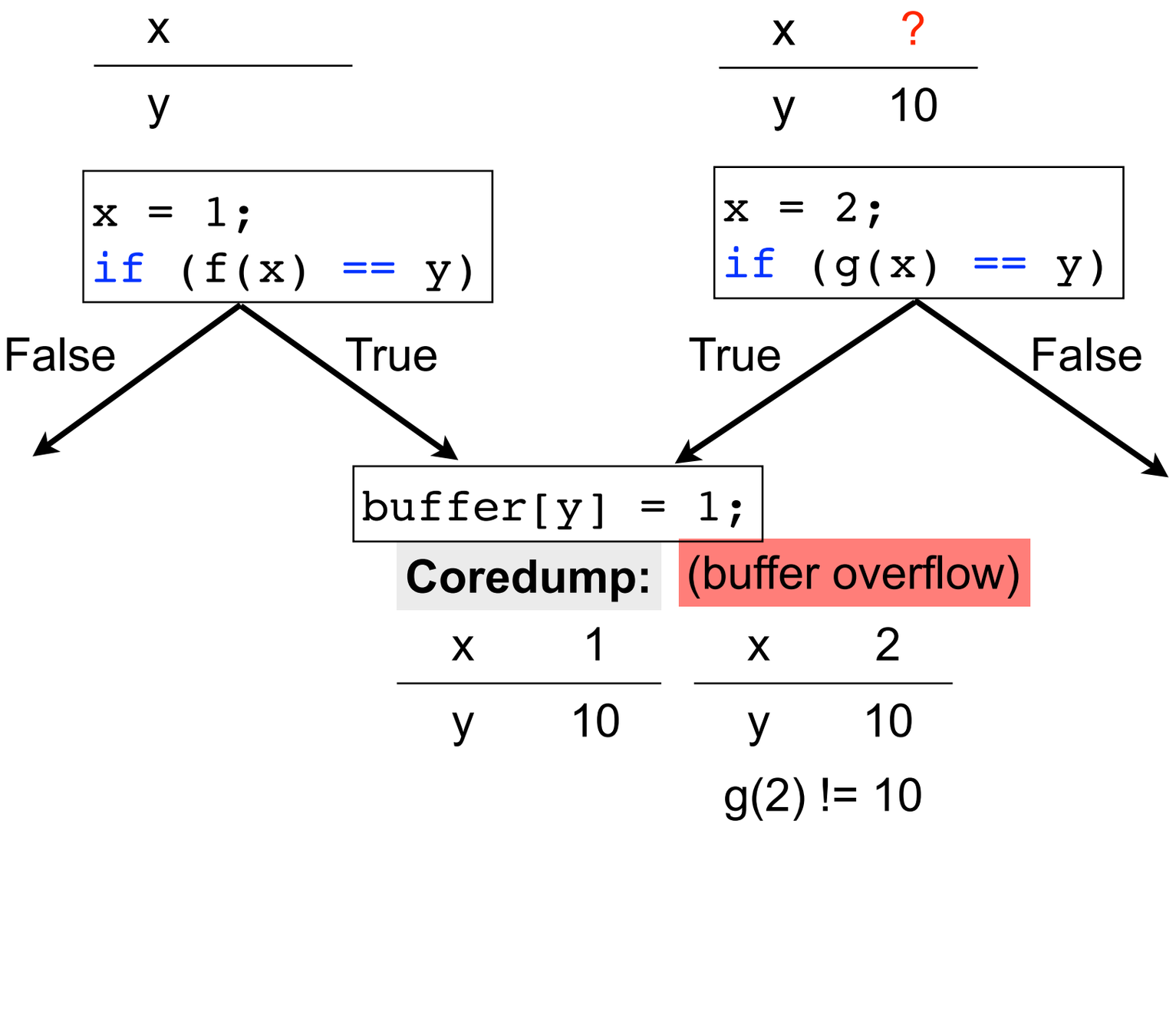
x	1	x	2
y	10	y	10

g(2) != 10

no match

Reverse Execution Synthesis

```
x = 1;
if (f(x) == y) {
  goto next;
}
...
x = 2;
if (g(x) == y) {
  goto next;
}
exit();
next:
  buffer[y] = 1
```



Reverse Execution Synthesis

```
x = 1;
if (f(x) == y) {
  goto next;
}
...
x = 2;
if (g(x) == y) {
  goto next;
}
exit();
next:
  buffer[y] = 1
```

x	
y	

x	
y	

```
x = 1;
if (f(x) == y)
```

False → | → True

```
buffer[y] = 1;
```

x	?
y	10

```
x = 2;
if (g(x) == y)
```

True → | → False

Coredump: (buffer overflow)

x	1
y	10

x	2
y	10

g(2) != 10

Reverse Execution Synthesis

```
x = 1;
if (f(x) == y) {
  goto next;
}
...
x = 2;
if (g(x) == y) {
  goto next;
}
exit();
next:
  buffer[y] = 1
```

x
y

```
x = 1;
if (f(x) == y)
```

False: x, y

True: x, ?, y, 10

```
x = 2;
if (g(x) == y)
```

True: x, y, 10

False

```
buffer[y] = 1;
```

Coredump: (buffer overflow)

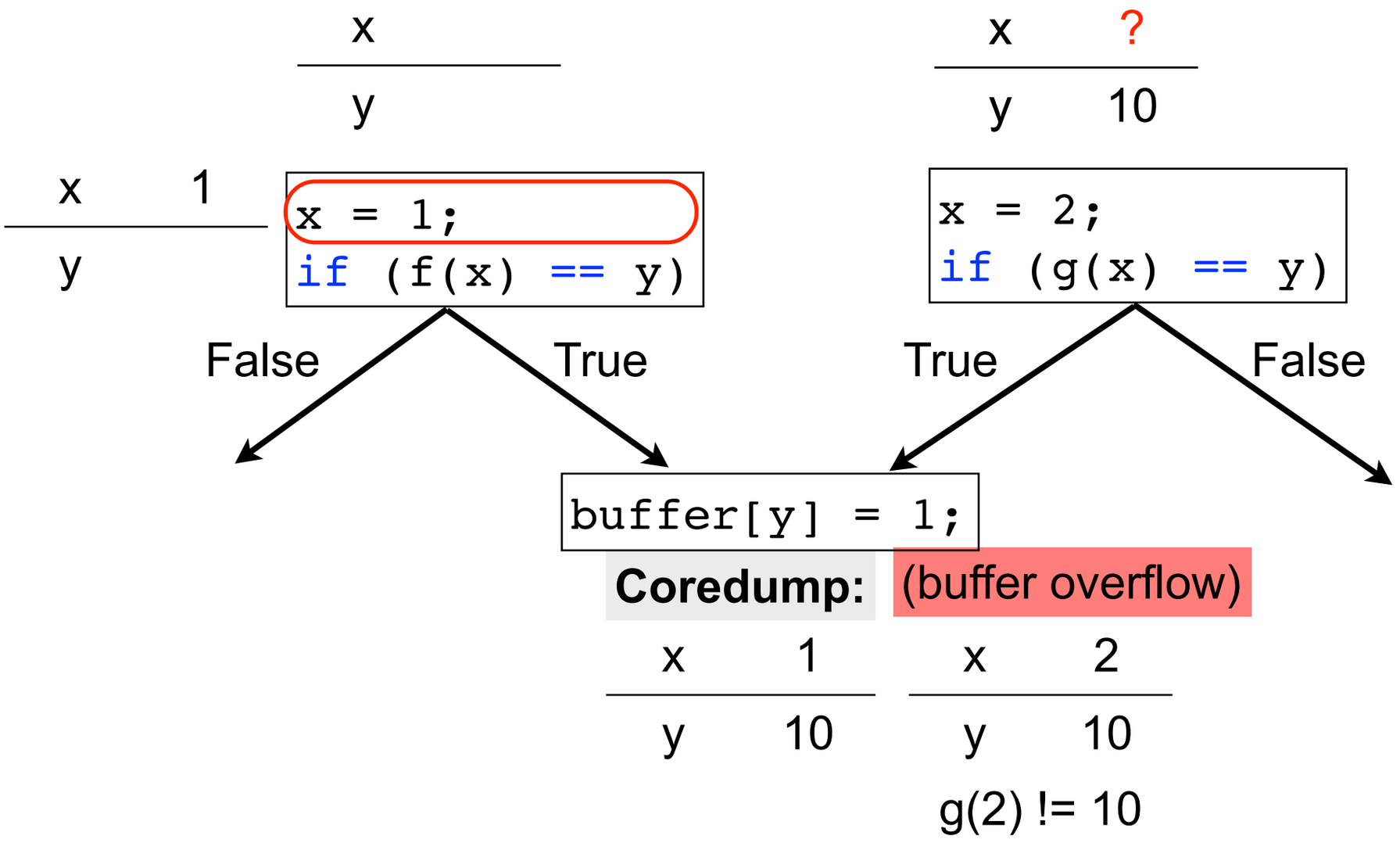
x, 1, y, 10

x, 2, y, 10

g(2) != 10

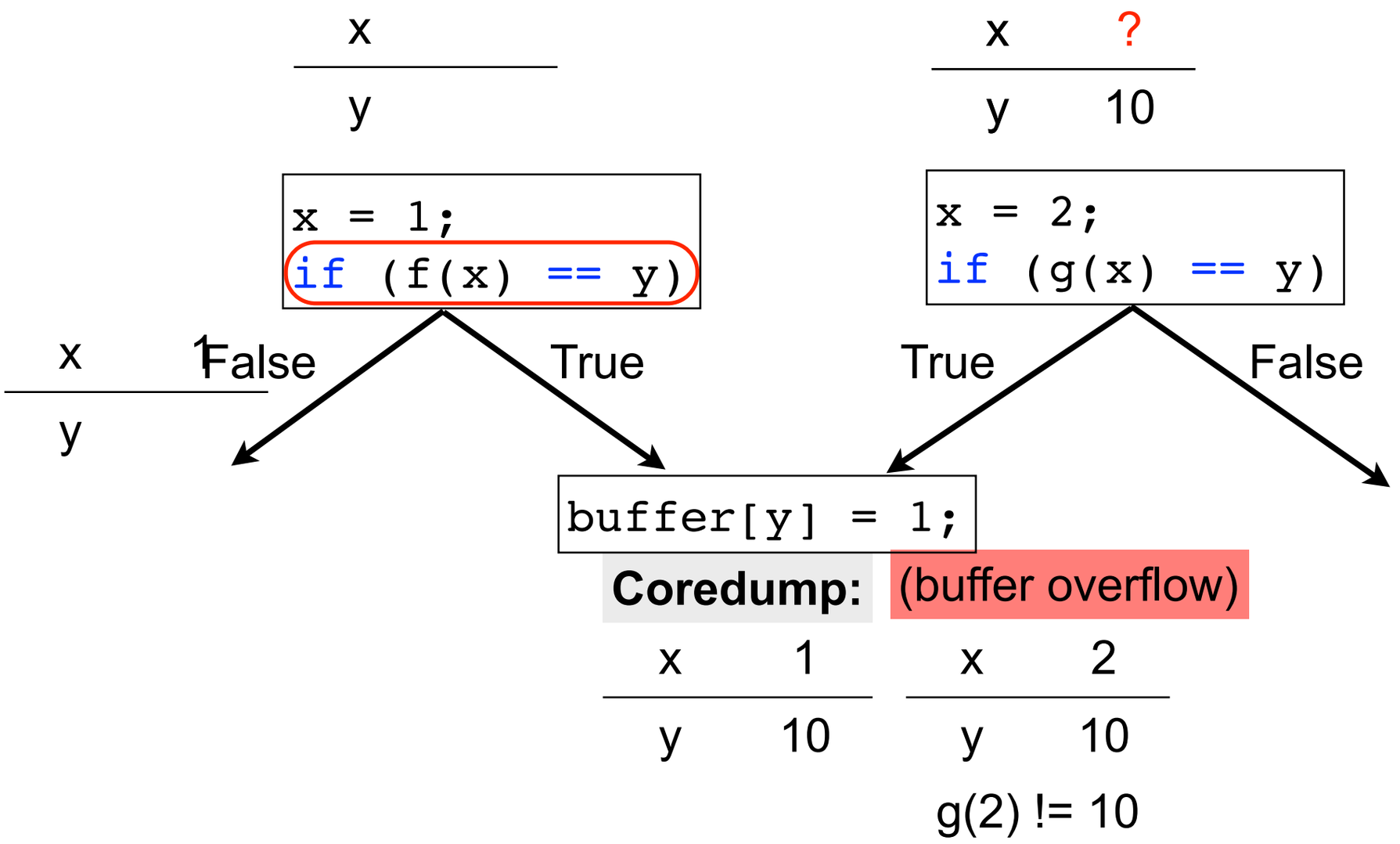
Reverse Execution Synthesis

```
x = 1;
if (f(x) == y) {
  goto next;
}
...
x = 2;
if (g(x) == y) {
  goto next;
}
exit();
next:
  buffer[y] = 1
```



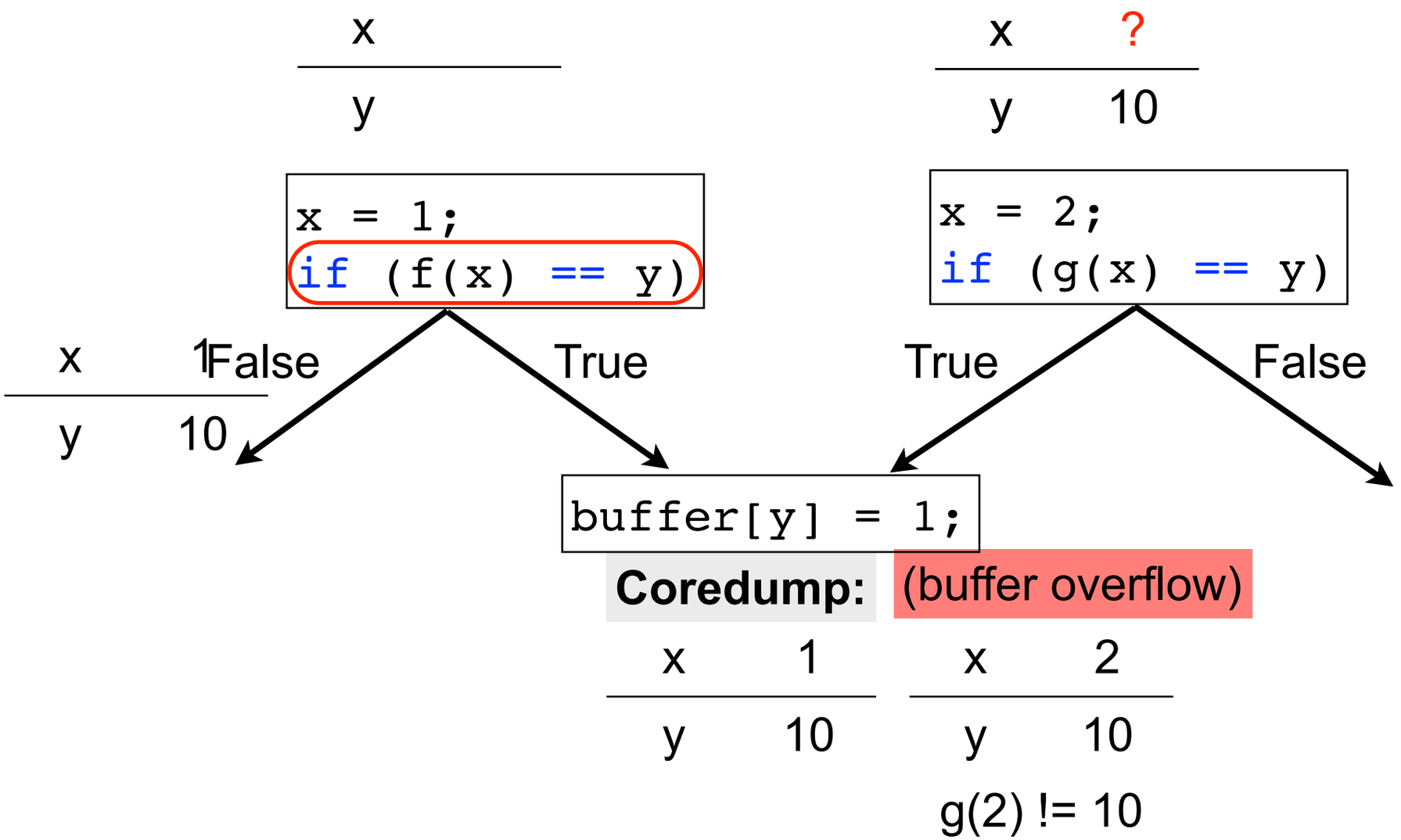
Reverse Execution Synthesis

```
x = 1;
if (f(x) == y) {
  goto next;
}
...
x = 2;
if (g(x) == y) {
  goto next;
}
exit();
next:
  buffer[y] = 1
```



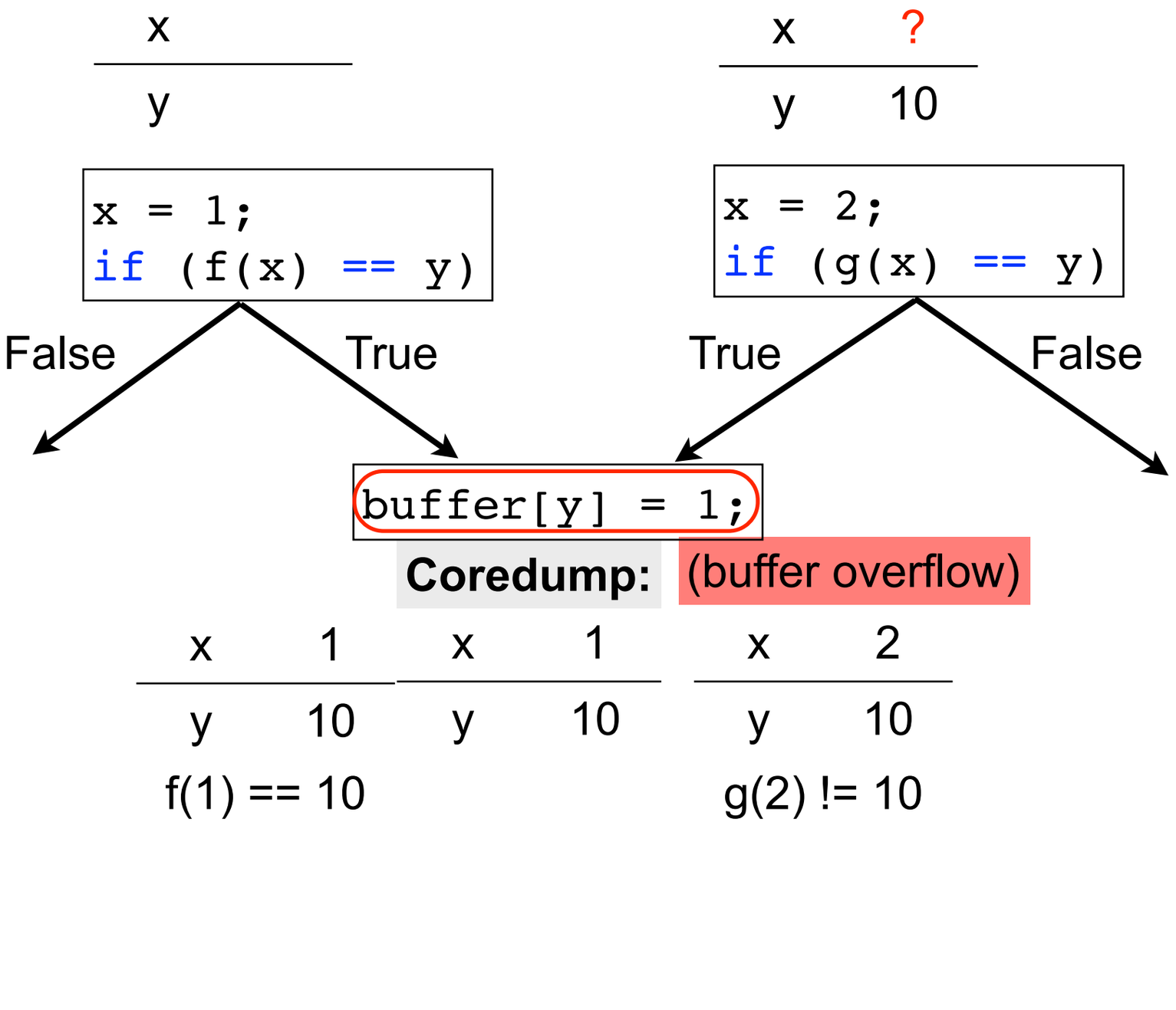
Reverse Execution Synthesis

```
x = 1;
if (f(x) == y) {
  goto next;
}
...
x = 2;
if (g(x) == y) {
  goto next;
}
exit();
next:
  buffer[y] = 1
```



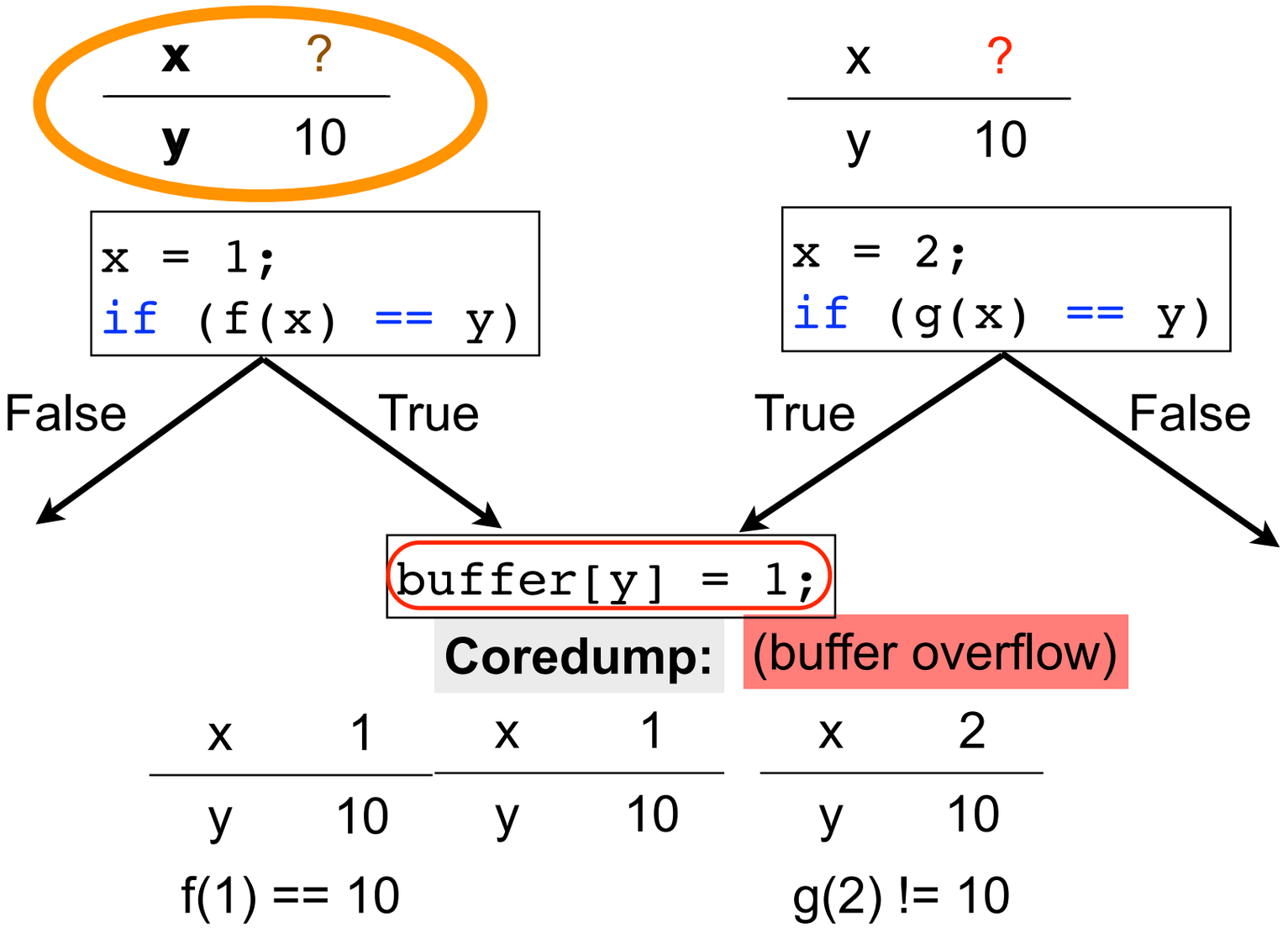
Reverse Execution Synthesis

```
x = 1;
if (f(x) == y) {
  goto next;
}
...
x = 2;
if (g(x) == y) {
  goto next;
}
exit();
next:
  buffer[y] = 1
```



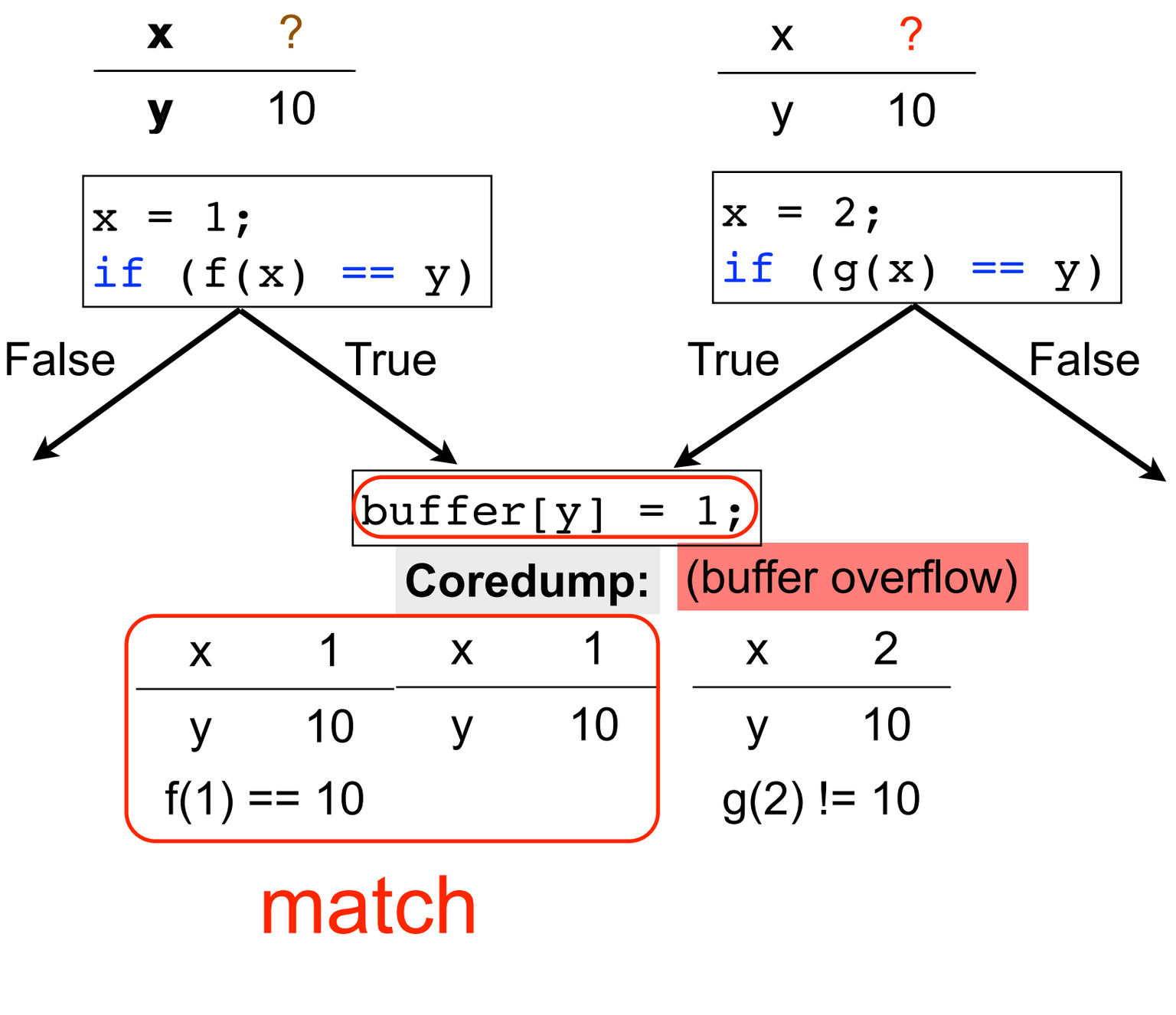
Reverse Execution Synthesis

```
x = 1;
if (f(x) == y) {
  goto next;
}
...
x = 2;
if (g(x) == y) {
  goto next;
}
exit();
next:
  buffer[y] = 1
```



Reverse Execution Synthesis

```
x = 1;
if (f(x) == y) {
  goto next;
}
...
x = 2;
if (g(x) == y) {
  goto next;
}
exit();
next:
  buffer[y] = 1
```



Reverse Execution Synthesis

```
x = 1;
if (f(x) == y) {
  goto next;
}
...
x = 2;
if (g(x) == y) {
  goto next;
}
exit();
next:
  buffer[y] = 1
```

Execution suffix

x	?
y	10

```
x = 1;
if (f(x) == y)
```

x	?
y	10

```
x = 2;
if (g(x) == y)
```

False True True False

```
buffer[y] = 1;
```

Coredump: (buffer overflow)

x	1	x	1
y	10	y	10

f(1) == 10

x	2
y	10

g(2) != 10

match

Reverse Execution Synthesis

```
x = 1;
if (f(x) == y) {
  goto next;
}
...
x = 2;
if (g(x) == y) {
  goto next;
}
exit();
next:
  buffer[y] = 1
```

Execution suffix

x	?
y	10

```
x = 1;
if (f(x) == y)
```

False | True

x	?
y	10

```
x = 2;
if (g(x) == y)
```

True | False

```
buffer[y] = 1;
```

Coredump: (buffer overflow)

x	1	x	1
y	10	y	10

f(1) == 10

match

x	2
y	10

g(2) != 10

```
(gdb) reverse-step
```

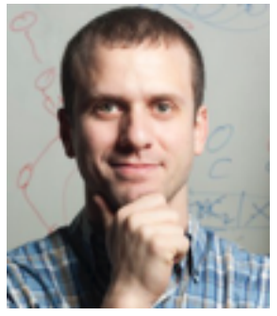
Reverse Execution Synthesis

- Coredump + program → execution suffix
- Debug arbitrarily long executions
- No runtime recording

Use Cases

- Automated debugging
  - *identify the root cause of a failure*
- Automated bug triaging
  - *triage based on the execution suffix*
- Identify likely hardware errors
  - *when no execution suffix explains the coredump*

Baris



Ed

George