Canonicalizing Execution for Automatic Debugging

Xiangyu Zhang

PURDUE
UNIVERSITY

December 16, 2011 @ ISHCS, PKU

Students

- William Nick Sumner (expected May 2012) and Bin Xin (Google)

Problem Statement

- Given a software runtime failure, we aim to automatically compute the explanation of the failure
  - A chain of execution steps that are causally correlated, leading from the root cause to the failure, called the causal path of the failure

A Conceptual Example

```
inventory = [(Shoes,5); (Hats,0); (Ties,1)]
bought = 0

for (item, available) in inventory:
  if bought < 3 and available >= 0:
    buy(item)
    bought += 1

print “Items bought: “, bought
```

Should print “Items bought: 2”

Failure: prints “Items bought: 3”

The Causal Path

A faulty branch is taken at A;

so bt is given the faulty value 2 at B;

so bt is given the faulty value 3 at C;

so '3' is printed erroneously at D.

```
1) for (itm, av) = (S,5):
2)   if bt < 3 and av >= 0:
3)       buy(itm)
4)       bt += 1
1) for (itm, av) = (H,0):
2)   if bt < 3 and av >= 0:      A
3)     buy(itm)
4)     bt += 1                   B
1) for (itm, av) = (T,1):
2)   if bt < 3 and av >= 0:
3)     buy(itm)
4)     bt += 1

5)print bt                       D
```

1. Leading from the root cause to the failure;
2. Values along the path are faulty;
3. Steps are causally correlated.

The Basic Idea

- Acquire a passing execution as the reference
- Compare the states of the passing and failing runs to isolate the failure causal path
- Delta Debugging (by Andreas Zeller) was proposed in 2002 to compute failure causal path
  - Our algorithm is completely different
  - We solved a number of key challenges for practicality

so '3' is printed erroneously at D.

Failing

```
1)for av = 5:
2)  if ... and av >= 0:
3)    buy(item)
4)    bt += 1 //=1
1)for av = 0:
2)  if ... and av >= 0:
3)    buy(item)
4)    bt += 1 //=2
1)for av = 1:
2)  if ... and av >= 0:
3)    buy(item)
4)    bt += 1  //=3
5)print  bt
```

D

Correct (patched)

```
1)for av = 5:
2)  if ... and av >= 0:
3)    buy(item)
4)    bt += 1  //=1
1)for av = 0:
2)  if !(... and av >= 0):


1)for av = 1:
2)  if ... and av >= 0:
3)    buy(item)
4)    bt += 1   //=2
5)print bt
```

3 is printed vs. 2 is printed

so bt is given the faulty value 3 at C ;
so '3' is printed erroneously at D .

Failing

```
1)for av = 5:
2)  if ... and av >= 0:
3)    buy(item)
4)    bt += 1 //=1
1)for av = 0:
2)  if ... and av >= 0:
3)    buy(item)
4)    bt += 1 //=2
1)for av = 1:
2)  if ... and av >= 0:
3)    buy(item)
4)    bt += 1  //=3
5)print  bt
```

C
D

Correct (patched)

```
1)for av = 5:
2)  if ... and av >= 0:
3)    buy(item)
4)    bt += 1  //=1
1)for av = 0:
2)  if !(... and av >= 0):

1)for av = 1:
2)  if ... and av >= 0:
3)    buy(item)
4)    bt += 1   //=2
5)print bt
```

bt =3   vs.   bt=2

so bt is given the faulty value 2 at B;
so bt is given the faulty value 3 at C ;
so '3' is printed erroneously at D .

Failing

```
  1)for av = 5:
  2)  if ... and av >= 0:
  3)    buy(item)
  4)    bt += 1 //=1
  1)for av = 0:
  2)  if ... and av >= 0:
  3)    buy(item)
B 4)    bt += 1 //=2
  1)for av = 1:
  2)  if ... and av >= 0:
  3)    buy(item)
C 4)    bt += 1  //=3
D 5)print  bt
```

Correct (patched)

```
1)for av = 5:
2)  if ... and av >= 0:
3)    buy(item)
4)    bt += 1  //=1
1)for av = 0:
2)  if !(... and av >= 0):

1)for av = 1:
2)  if ... and av >= 0:
3)    buy(item)
4)    bt += 1   //=2
5)print bt
```

bt =2 vs. bt=1

A faulty branch is taken at A;
so bt is given the faulty value 2 at B;
so bt is given the faulty value 3 at C;
so '3' is printed erroneously at D.

Failing

```
  1)for av = 5:
  2)  if ... and av >= 0:
  3)    buy(item)
  4)    bt += 1 //=1
  1)for av = 0:
A 2)  if ... and av >= 0:
  3)    buy(item)
B 4)    bt += 1 //=2
  1)for av = 1:
  2)  if ... and av >= 0:
  3)    buy(item)
C 4)    bt += 1  //=3
D 5)print  bt
```

Correct (patched)

```
1)for av = 5:
2)  if ... and av >= 0:
3)    buy(item)
4)    bt += 1  //=1
1)for av = 0:
2)  if !(... and av >= 0):

1)for av = 1:
2)  if ... and av >= 0:
3)    buy(item)
4)    bt += 1   //=2
5)print bt
```

(... and av >= 0) is True vs. **!(... and av >= 0) is False**

Algorithm Overview

1 align (E1, E2) — Execution alignment
2 FOR each aligned point n in the reverse execution order DO
3 re-execute both E1 and E2 to n
4 take memory snapshots M1 and M2, of E1 and E2, resp.
5 canonicalize M1 and M2 — Memory canonicalization
6 Δ = state differences between M1 and M2
7 report the minimal subset of Δ critical to the failure — Causality testing
8 ENDFOR

include complex data structures and pointer values

Outline

- Execution alignment
- Canonicalizing memory
- Causality testing
- Results

Outline

- Execution alignment
- Canonicalizing values
- Causality testing
- Results

Execution Alignment

- Informal problem statement
  - Given an execution point in a run, what is the corresponding point in another run
- Current practice
  - statement (+ instance)

Problematic in the presence of control flow differences, loops, and recursion.

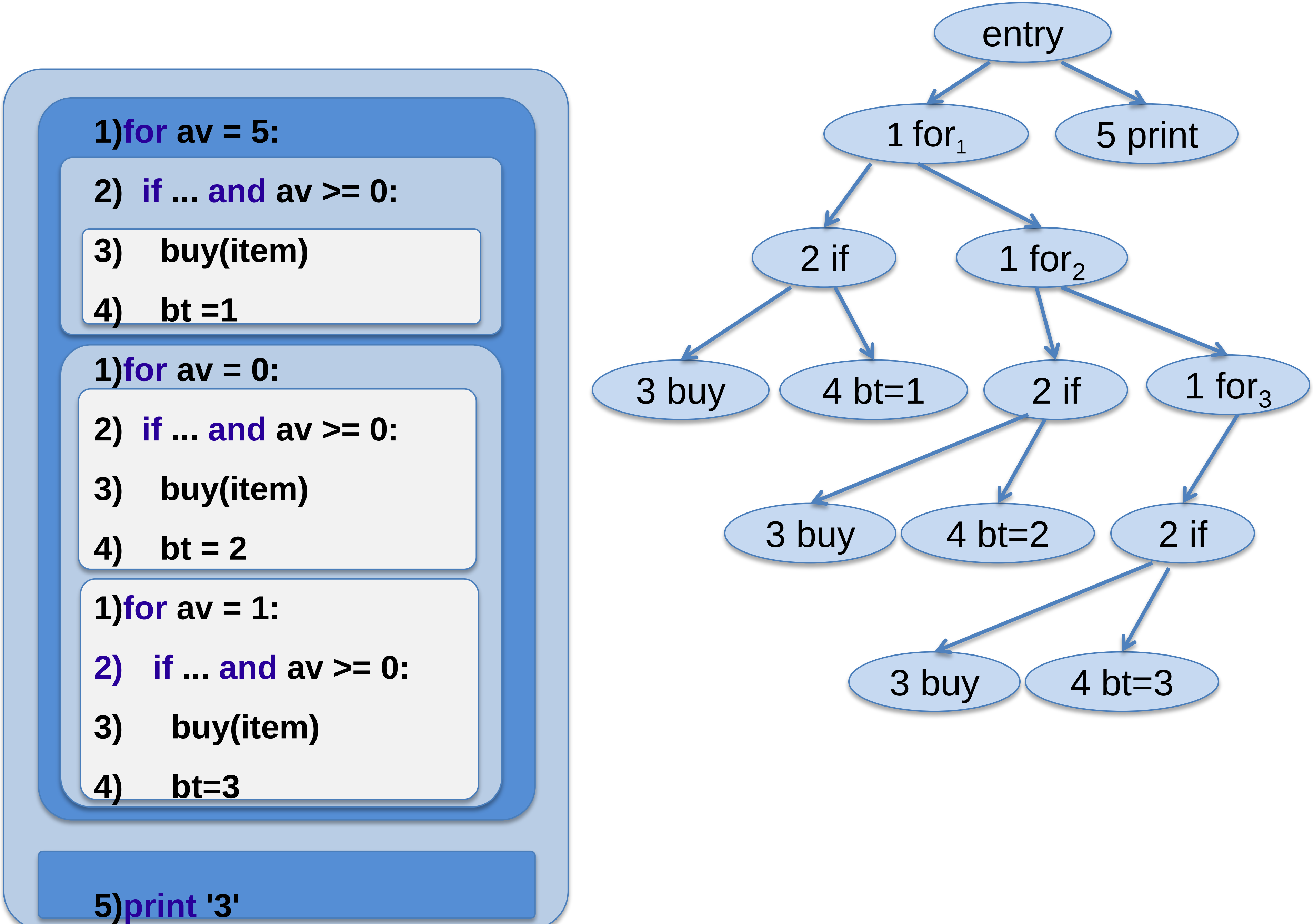
Instance Count Does NOT work

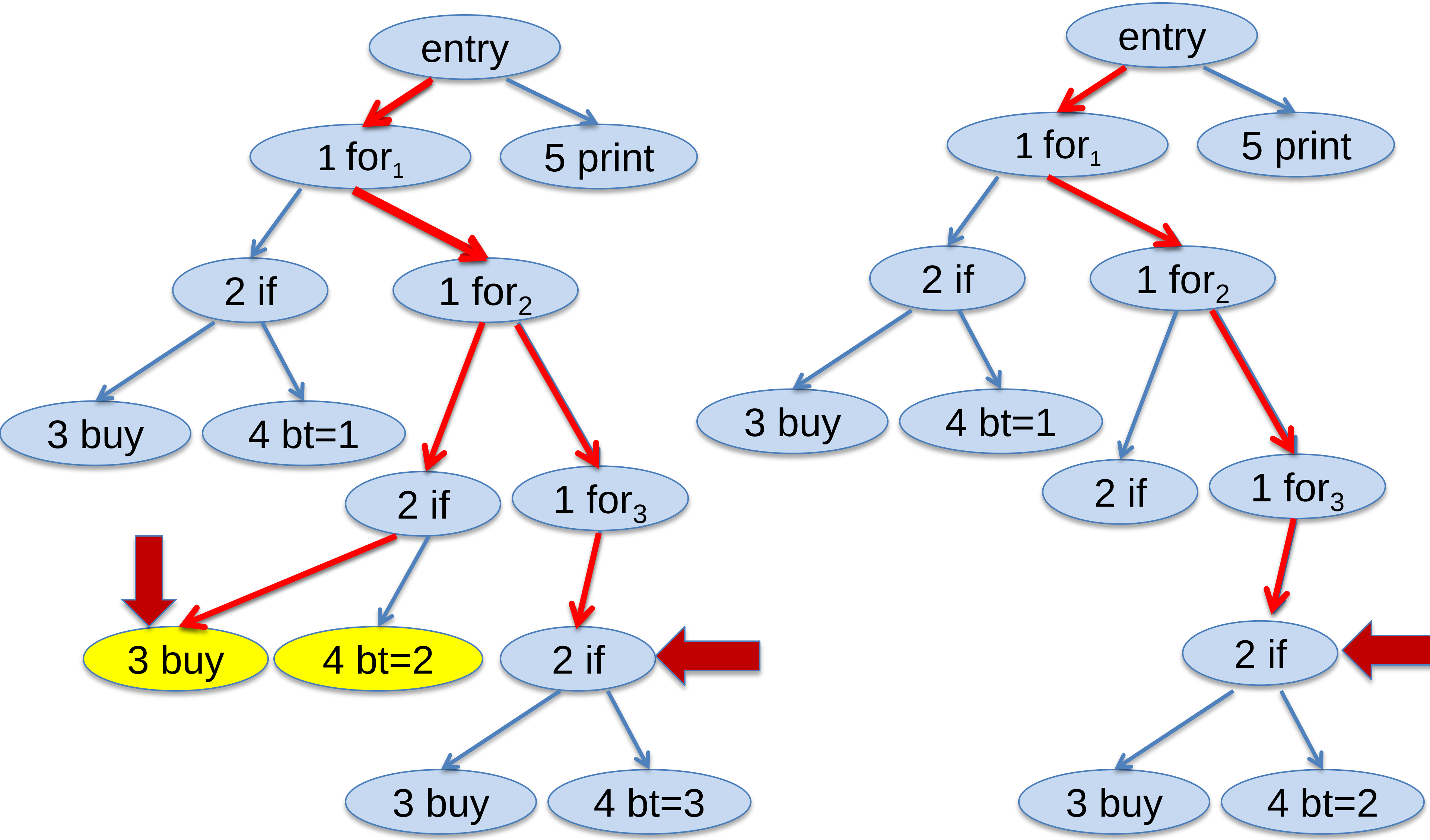
Failing	Correct (patched)
1)for ... 2) if ... and av >= 0: 4) bt = 1	1)for ... 2) if ... and av >= 0: 4) bt = 1
1)for ... 2) if ... and av >= 0: 4) bt = 2	1)for ... 2) if !(... and av >= 0):
1)for ... 2) if ... and av >= 0: 4) bt = 3	1)for ... 2) if ... and av >= 0: 4) bt = 2
5)print bt	5)print bt

bt =2 vs. bt=2

Our Solution – Execution Indexing (PLDI'08)

- Concept
  - Develop a canonical representation, called the index, of an execution point. Points with the same index across runs align
- Basic idea
  - An execution is modeled as a trace. The trace is parsed into an index tree that represents its nesting structure. Indices are defined based on the tree.


1)for av = 5:
2) if ... and av >= 0:
3) buy(item)
4) bt =1
1)for av = 0:
2) if ... and av >= 0:
3) buy(item)
4) bt = 2
1)for av = 1:
2) if ... and av >= 0:
3) buy(item)
4) bt=3
5)print '3'
entry
1 for1
5 print
2 if
1 for2
3 buy
4 bt=1
2 if
1 for3
3 buy
4 bt=2
2 if
3 buy
4 bt=3



The index of an execution point is the path leading from the root to that point

[entry, 1, 1, 2, 3]

Points with the same index align

Indexing

- We don’t actually collect trace
- We compute the index of the current execution point on the fly
  - In order to make use of the online algorithm.
    - At a point of interest, issue the query “acquire the current index X” (e.g. X=[entry, 1, 1, 1, 2])
    - In the other run, issue “execute to the index X”.

The Indexing Algorithm

- Use a stack to represent the current index
- Push on a predicate, a function entry
- Pop on the immediate post-dominator or function exit.

[entry]	**Entry**
[entry, 1]	**1)for ...**
[entry, 1, 2]	**2) if ... and av >= 0:**
[entry, 1, 2]	**4) bt = 1**
[entry, 1, 1]	**1)for ...**
	2) if ... and av >= 0:
	4) bt = 2
	1)for ...
	2) if ... and av >= 0:
	5)print bt

Indexing Algorithm Continued

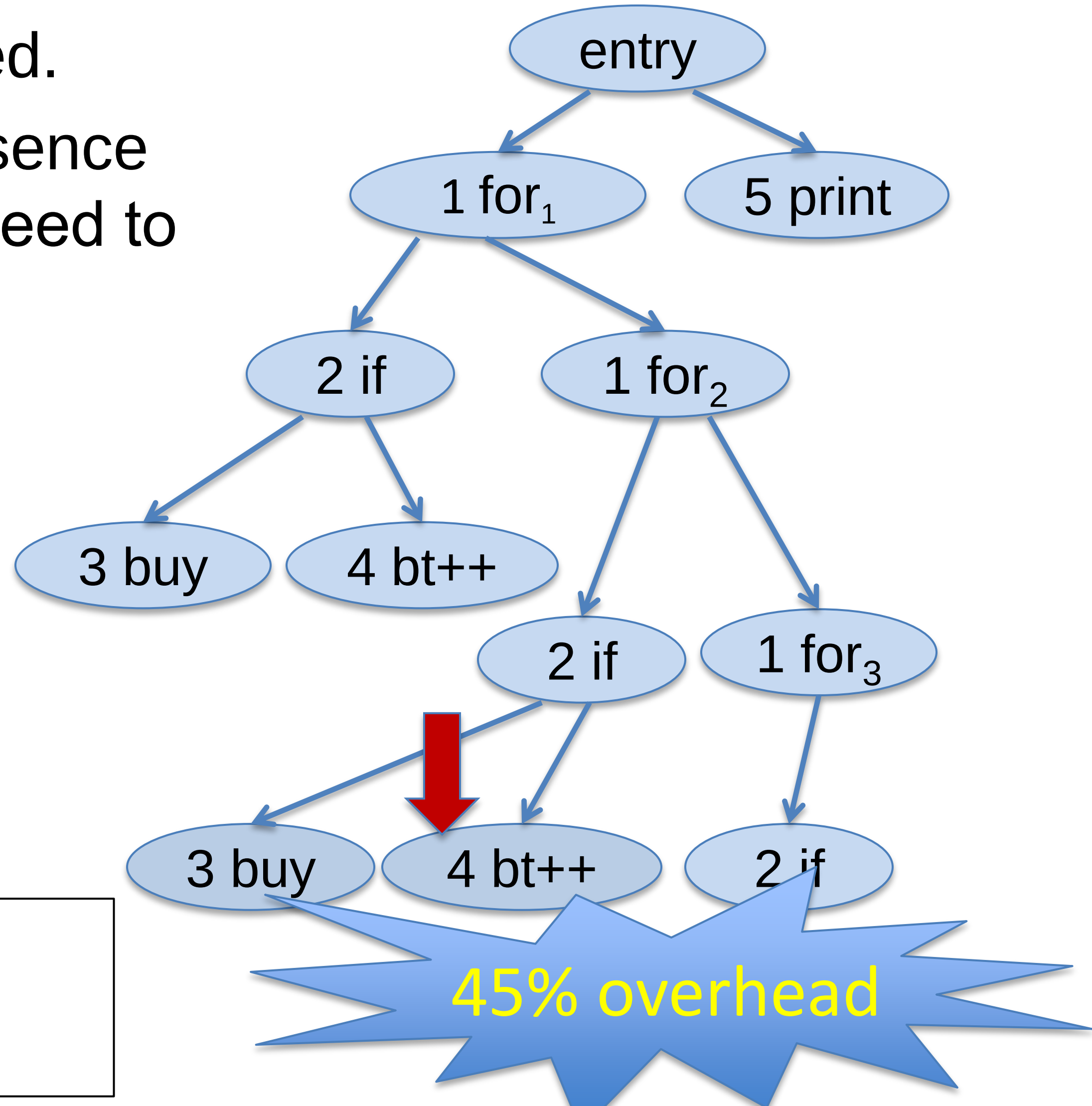
- Iterations are coalesced.
- Predicates whose presence can be inferred don't need to be pushed

[entry, 1, 1, 2, 4]

[entry, 1:2, 2, 4]

[entry, 1:2, 4]

```
1)for ...
2)  if ... and av >= 0:
4)      bt += 1
```

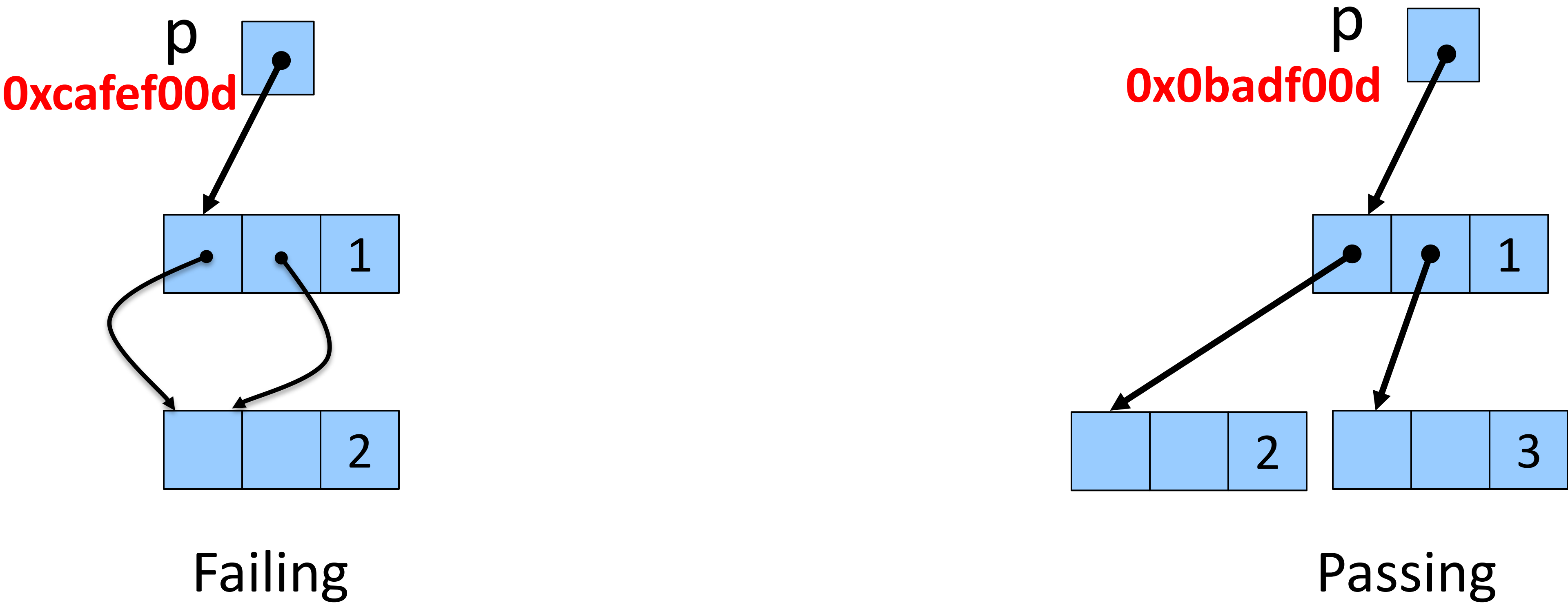


Outline

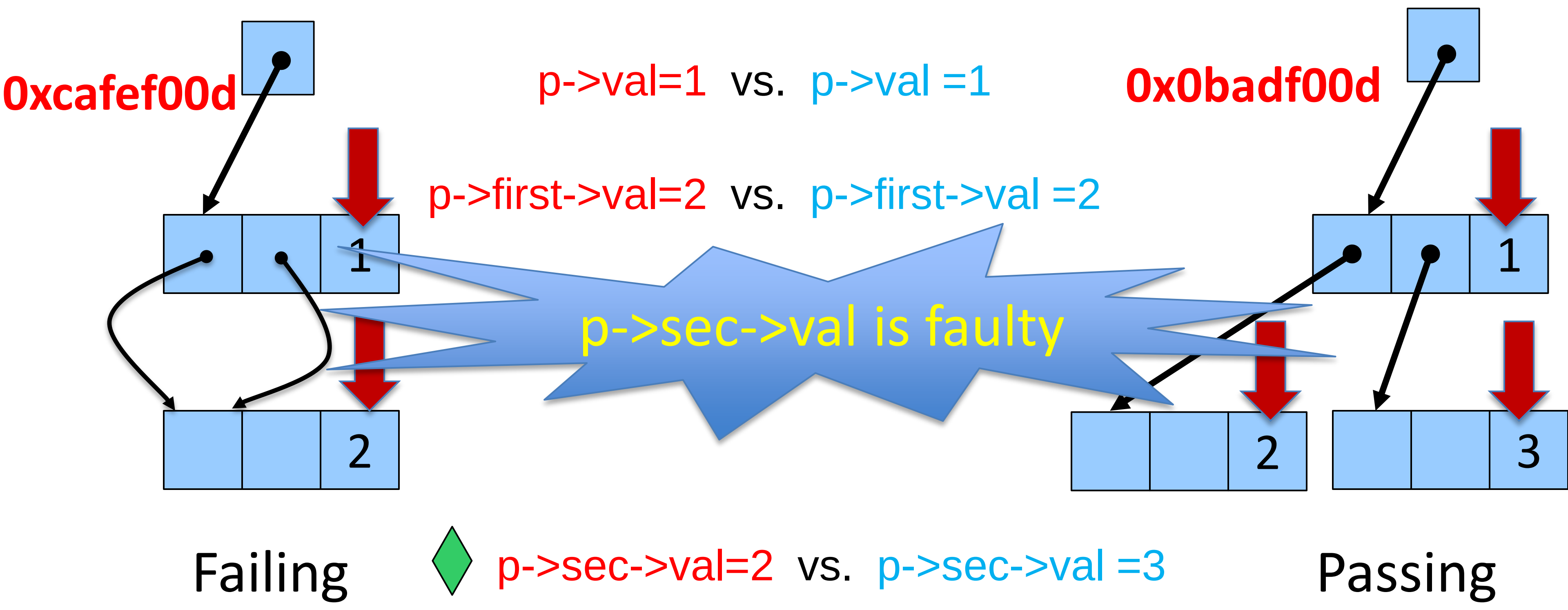
- Execution alignment [PLDI’08]
- Canonicalizing memory [FSE’10]
- Causality testing
- Getting the passing run
- Results

Canonicalizing Memory

- Informal problem statement
  - The same heap object may be present in different locations in the two runs
  - Pointer values are not directly comparable



- Solution: heap object correspondence is established by symbolic reference path
  - imprecision due to aliasing

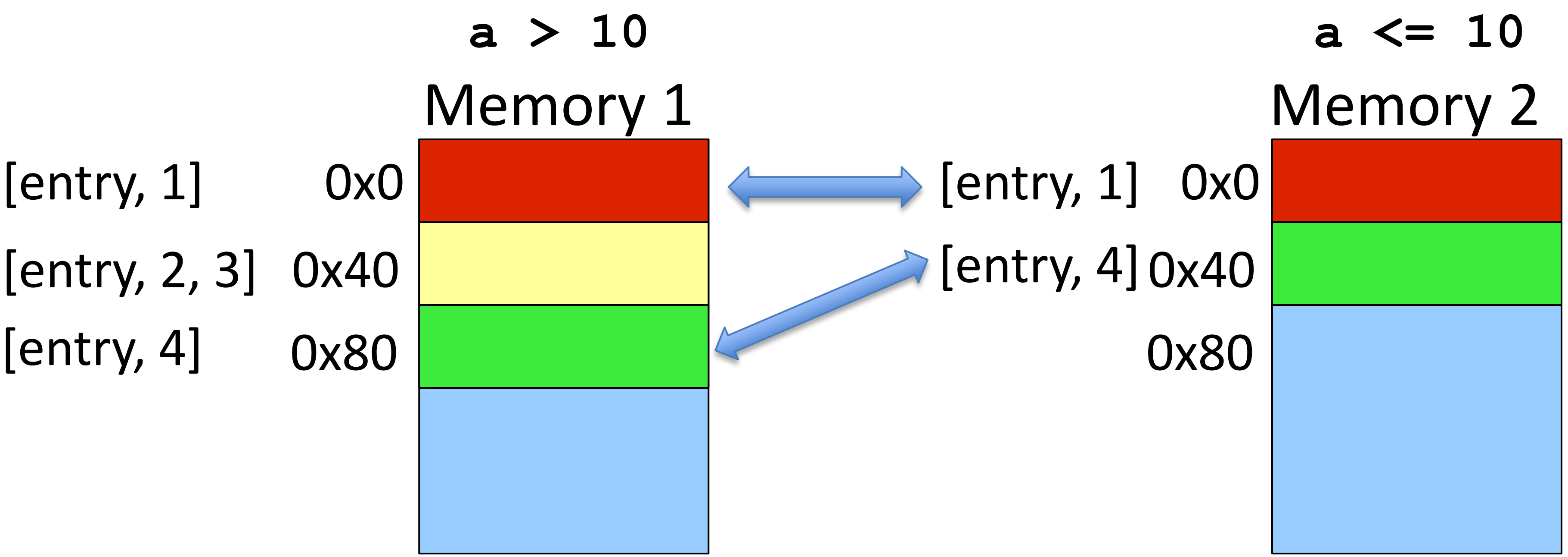


Our solution – Memory Indexing

- Concept
  - Develop a canonical representation, called the memory index, for each allocated heap location. Locations with the same memory index across runs align
- Basic idea
  - An allocated heap object is denoted by the control flow index of its dynamic allocation point
    - Memory alignment is reduced to control flow alignment
  - A precise, dynamic analogue to static heap abstraction

A Conceptual Example

```
1) x = malloc(64)
2) if (a > 10)
3)    y = malloc(64)
4) z = malloc(64)
```



Online Algorithm – Flow Semantics

Execution	Action
p=malloc (…)	mIdx[p]=(current_cfIdx,0)
q= p + c	mIdx[q]= (mIdx[p].first, mIdx[p].second+ c)

```
1  if (…)
2    if (…)
3       p= malloc(64)
4  q = p + 10
```

mIdx[p]=([entry,1,2,3], 0)

mIdx[q]=([entry,1,2,3], 10)

Outline

- Execution alignment [PLDI'08]
- Canonicalizing memory [FSE'10]
- Causality testing
- Results

Causality Testing

- Informal problem statement
  - At aligned points, determine whether a subset of state differences causes the failure, by testing whether copying the subset from the failing run to the passing run induces the failure
- Challenges
  - Copying state from one execution to another in a robust way
- Basic idea
  - Concrete memory => canonical memory
  - Compare canonical memory
  - Copy canonical differences from the failing to the pass
  - Concretize these differences in the passing run
  - Resume execution

Outline

- Execution alignment [PLDI'08]
- Canonicalizing memory [FSE'10]
- Causality testing
- Results

Implementation

- LLVM, Python, and C++
  - >10K LOC
- Support programs in C/C++
  - as large as gcc

Effectiveness in Auto-Debugging

Program	Bug ID	Passing	Max Diff (w. MI)	Max Diff (w/o MI)	Step/ Diff	Time (s)
gcc-2.95	529	patching	233	8365	8/1	4559
gcc-2.95	776	patching	230	10101	2/1	379
gcc-2.95	2771	diff input	284	11095	4/1	1797
make 3.81	16958	non-reg	184	2699	9/1	740
make 3.81	18435	diff input	336	3301	29/2	645
make 3.81	19133	diff input	550	3728	5/2	235
make 3.80	112	diff input	645	3309	11/6	685
gawk 3.1	2006	diff input	22	630	8/1	56

- Canonicalization is very critical
- Computed causal paths are short and thin
- Computation time is reasonable
- Other over 20 real bugs for grep, gzip, tar, and flex.

Recent Results

- Finishing validating our technique on a very popular debugging test-bed – Siemens suite
    - >10,000 runtime failures
    - Precisely capture the root causes
    - Present the precise explanations for all those failures online

Demo

http://www.youtube.com/watch?v=36cXwVqMo30

Related Work

- Delta debugging [Zeller, FSE '02]
- Statistical debugging [Liblit, PLDI'05]
- Dynamic slicing [Zhang, PLDI'04]
- Trace based correspondence [Ramanathan, FASE '06]
- Others…

Conclusions

- We develop three important primitives
  - Control flow indexing
  - Memory indexing
  - Causality testing
- An effective auto-debugging engine can be composed from the primitives
  - Handles real bugs for non-trivial programs such as gcc
- Execution comparison needs canonicalization

Thank you!

http://www.cs.purdue.edu/homes/xyzhang