IT5440- NGUYÊN LÝ VÀ KỸ THUẬT PHÂN TÍCH CHƯƠNG TRÌNH

(Principle and Technique of Program Analysis)

AY 2025-2026

Giảng viên: PGS. TS.Huỳnh Quyết Thắng
Khoa Khoa học máy tính
Trường Công nghệ thông tin và Truyền thông
www.soict.hust.edu.vn/~thanghq

Chapter III. Dynamic Program analysis

- *Tracing*
- *Profiling*
- *Checkpointing and replay*
- *Dynamic slicing*
- *Execution indexing*
- *Memory Detection*
- *Fault localization*

Introduction

- Dynamic program analysis is to solve problems regarding software dependability and productivity by inspecting software execution.
- Program executions vs. programs
  - Not all statements are executed; one statement may be executed many times.
  - Analysis on a single path – the executed path
  - All variables are instantiated (solving the aliasing problem)

  Resulting in:
  - Relatively lower learning curve.
  - Precision.
  - Applicability.
  - Scalability.
- Dynamic program analysis can be constructed from a set of primitives
  - Tracing
  - Profiling
  - Checkpointing and replay
  - Dynamic slicing
  - Execution indexing
  - Delta debugging
- Applications
  - Dynamic information flow tracking
  - Automated debugging

What is Tracing

- Tracing is a process that faithfully records detailed information of program execution (lossless).
  - Control flow tracing
    - the sequence of executed statements.
  - Dependence tracing
    - the sequence of exercised dependences.
  - Value tracing
    - the sequence of values that are produced by each instruction.
  - Memory access tracing
    - the sequence of memory references during an execution
- The most basic primitive.

Why Tracing

- Debugging
  - Enables time travel to understand what has happened.
- Code optimizations
  - Identify hot program paths;
  - Data compression;
  - Value speculation;
  - Data locality that help cache design;
- Security
  - Malware analysis
- Testing
  - Coverage.

Outline

- What is tracing.
- Why tracing.
- How to trace.
- Reducing trace size.
- Trace accessibility

Tracing by Printf

```
Max = 0;
for (p = head; p; p = p->next)
{
   printf("In loop\n");
   if (p->value > max)
   {
      max = p->value;
   }  printf("True branch\n");
}
```

Tracing by Source Level Instrumentation

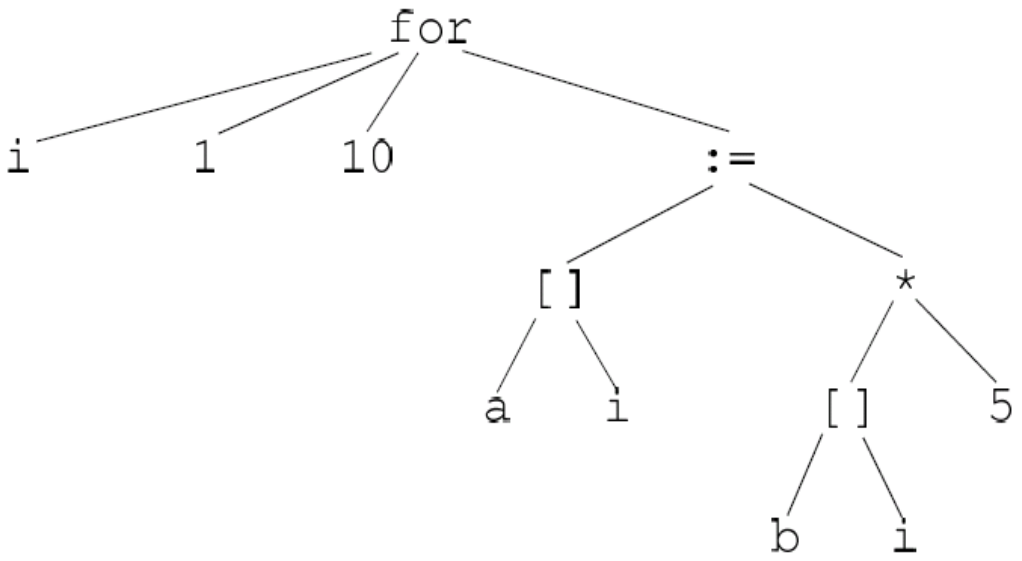
- Read a source file and parse it into ASTs.
- Annotate the parse trees with instrumentation.
- Translate the annotated trees to a new source file.
- Compile the new source.
- Execute the program and a trace produced.

An Example

Source:

```
for i := 1 to 10 do
  a[i] := b[i] * 5;
end
```

AST:

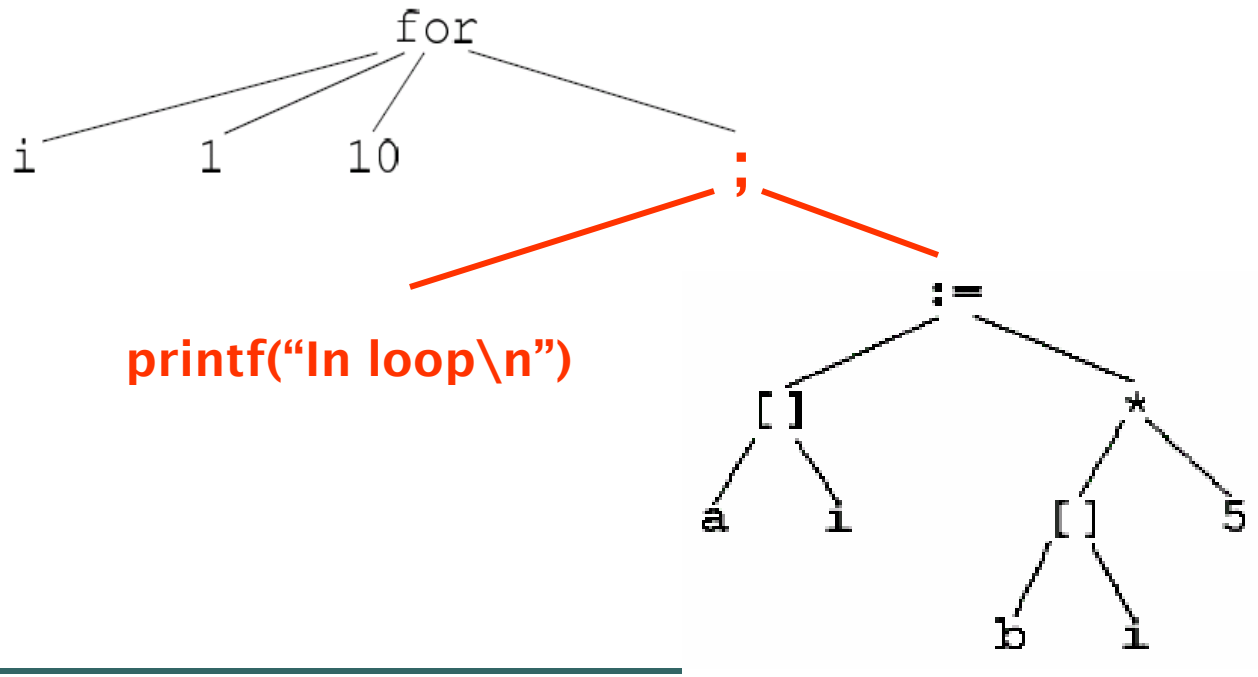


An Example

Source:

```
for i := 1 to 10 do
  a[i] := b[i] * 5;
end
```

AST:



Limitations of Source Level Instrumentation

- Hard to handle libraries.
  - Proprietary libraries: communication (MPI, PVM), linear algebra (NGA), database query (SQL libraries).
- Hard to handle multi-lingual programs
  - Source code level instrumentation is heavily language dependent.
- Requires source code
  - Worms and viruses are rarely provided with source code

Tracing by Binary Instrumentation

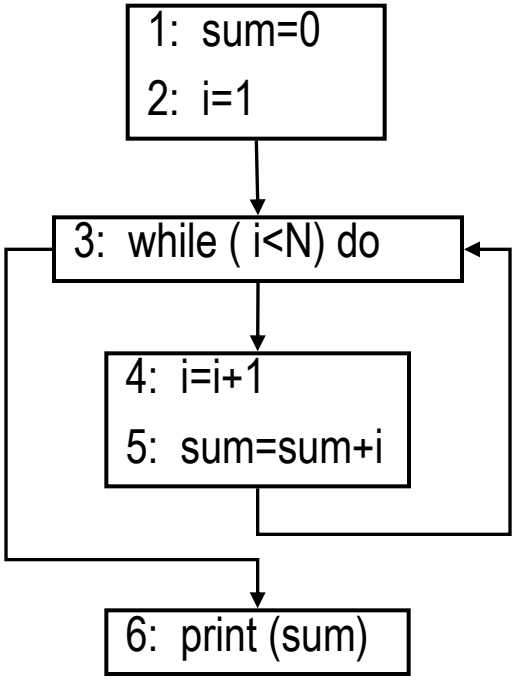
- What is binary instrumentation
  - Given a binary executable, parses it into intermediate representation. More advanced representations such as control flow graphs may also be generated.
  - Tracing instrumentation is added to the intermediate representation.
  - A lightweight compiler compiles the instrumented representation into a new executable.
- Features
  - No source code requirement
  - Easily handle libraries.

Outline

- What is tracing.
- Why tracing.
- How to trace.
- Reducing trace size.

Fine-Grained Tracing is Expensive

```
1:    sum=0
2:    i=1
3:    while ( i<N) do
4:          i=i+1
5:          sum=sum+i
      endwhile
6:  print(sum)
```

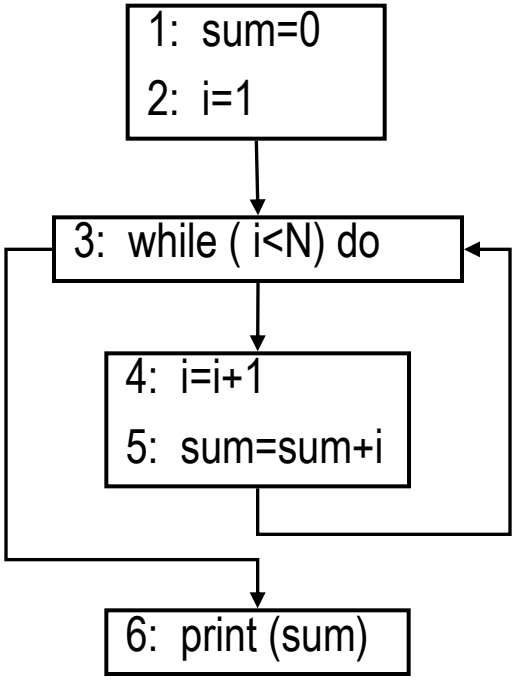


Trace(N=6): 1 2 3 4 5 3 4 5 3 4 5 3 4 5 3 4 5 3 6

Space Complexity: 4 bytes * Execution length

Basic Block Level Tracing

```
1:    sum=0
2:    i=1
3:    while ( i<N) do
4:          i=i+1
5:          sum=sum+i
      endwhile
6:  print(sum)
```



Trace(N=6): 1 2 3 4 5 3 4 5 3 4 5 3 4 5 3 4 5 3 6

BB Trace: 1 3 4 3 4 3 4 3 4 3 4 3 6

More Ideas

- Would a function level tracing idea work?
  - A trace entry is a function call with its parameters.
- Predicate tracing

```
1:    sum=0
2:    i=1
3:    while ( i<N) do
4:          i=i+1
5:          sum=sum+i
      endwhile
6:  print(sum)
```

Instruction trace	Predicate trace
1 2 3 6	F
1 2 3 4 5 3 6	T F

  - Lose random accessibility
- Path based tracing

Compression

- Using zlib
  - Zlib is a software library used for data compression. It wraps the compression algorithm used in gzip.
  - Divide traces into trunks, and then compress them with zlib.
  - Disadvantage: trace can only be accessed after complete decompression; slow
- Desired features
  - Accessing traces in their compressed form.
  - Traversing forwards and backwards.
  - fast

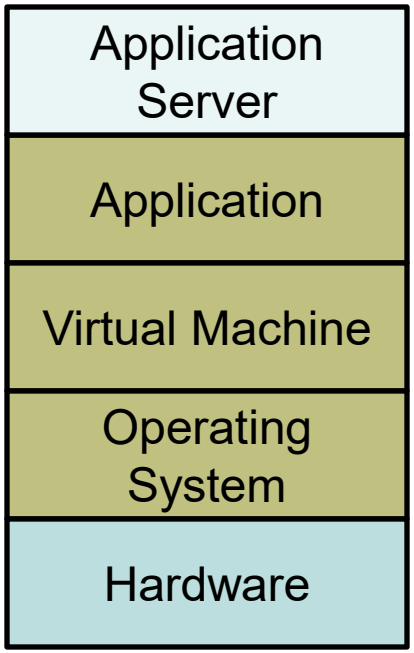
Profiling

- Profiling: Analysis of program behavior based on run-time data
- Profiler: conceptual module whose purpose is to collect or analyse runtime data.
- Profile: a set of frequencies associated with run-time events
- Profile usage examples:
  - Debugging and bug-isolation.
  - Feedback based optimization.
  - Coverage testing .
  - Understanding program/architecture interaction.

Profiling

- Qualities of a profiler desired:
  - Accurate
  - Low-overhead
  - Fast convergence
  - Flexible
  - Portable
  - Transparent
  - Low storage overhead

Motivation



Application:

- Number of features and usage scenarios.
- Large blocks of code

Hardware:

- Heterogeneous
- Pre-fetching, caching

Motivation

To capture runtime-behavior:

Static analysis:

- *fails. Why ?*
  - Software and Hardware are complex (solve complex problems).
  - Software-Hardware are difficult to comprehend.
  - Application usage scenario is hard to predict.

Dynamic analysis: based on run-time information

Types of profiles

- Point profile
  - *Events are simple*
  - *Events are independent*

Example: basic block, cache-misses etc.

- Context profile
  - *Events are composed of simpler ordered events*

Example: call-context : sequence of method invocations

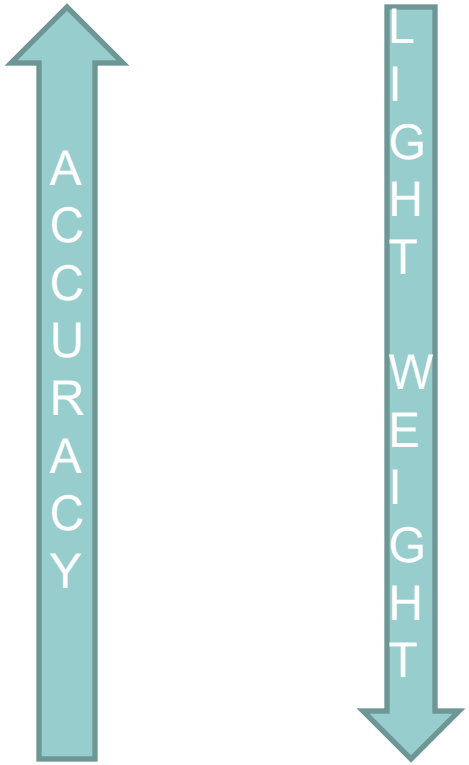
execution paths : sequence of edges in a CFG

Profiling techniques

Exhaustive instrumentation

Instrumentation sampling, temporary instrumentation, …

Sampling



Implementation of Profiler

• *Hybrid (HW-assisted)*

1. Hardware Performance Monitors (HPMs)
2. Dedicated HW collectors that deliver data to SW module

- Fixed, low-overhead

• *Software*

– Pure software implementations

- Portable, flexible, high-overhead

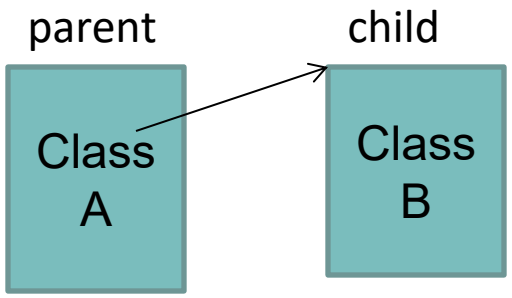
Hybrid profiling systems

- *HPM-based system-wide profilers*
  - *Event-based sampling*
  - *Support multiple events*
  - *Profile unmodified binaries*
  - *Low overhead*
  - *May be used in production environment*

Hybrid profiling systems

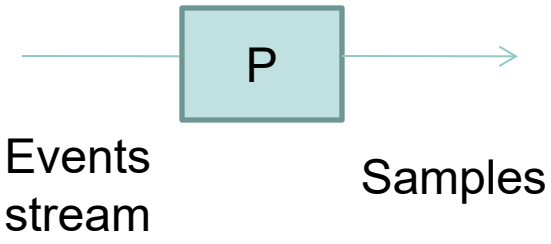
- HPM-sampling for dynamic compilation
  - HPM-sampling for JikesRVM
    - More accurate, converges faster, less overhead
  - Speed-up by 5 - 18%

- Online optimizations using HPMs
  - Cache-misses profiling
  - Co-allocation of objects
  - O.H. < 1%, speed-up 14%

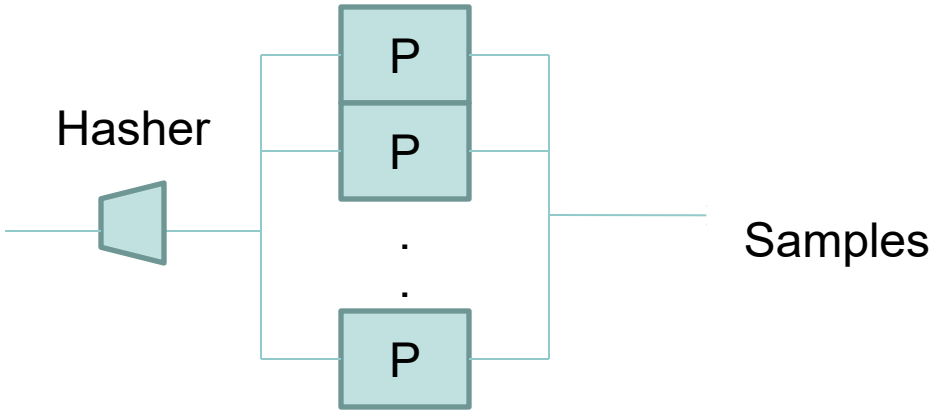


Hybrid profiling systems

- Rapid profiling via stratified sampling
  - Data stream is divided into disjoint strata
  - Reduces size of output stream and improves accuracy
  - O.H. 4.5%, accuracy 97%

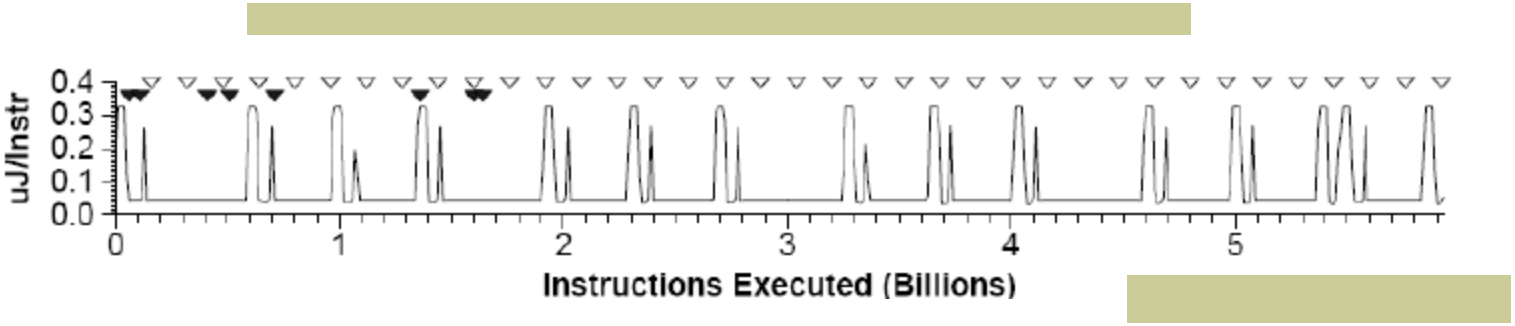


Periodic Sampler



Stratified Sampling

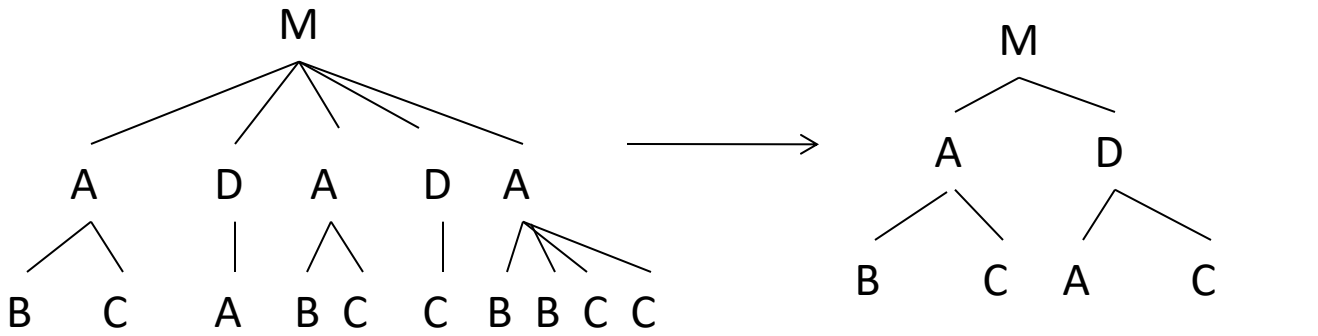
Hybrid profiling systems



- Phase-aware profiling
  - Programs exhibit repeating patterns of execution (phases)
  - Profiling a representative of each phase approximates full profile
  - Phase-change detector is in HW
  - O.H. reduction by 58% over periodic sampling, accuracy 95%

Software profiling systems

- Dynamic call tree
  - Fully describes method invocation during execution
- Calling context tree (CCT)
  - Aggregates calls with same context

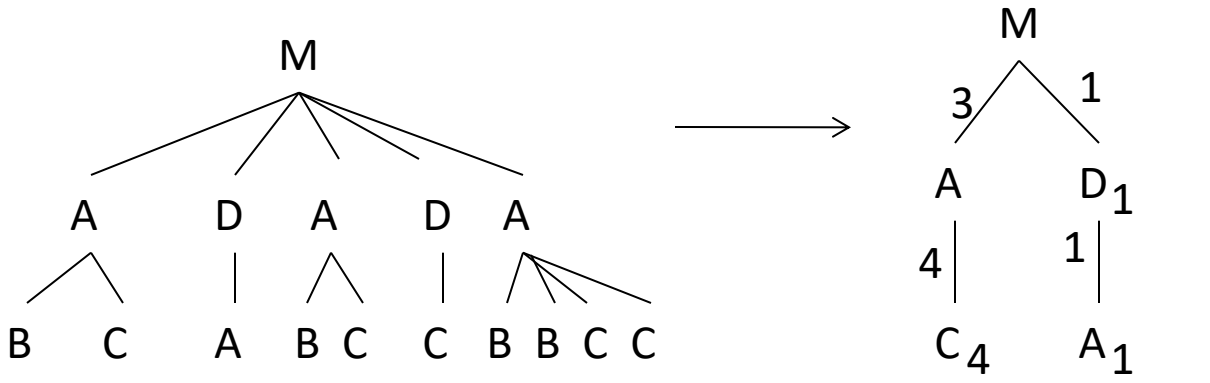


Dynamic call tree

Calling context tree

Software profiling systems

- Calling-context profiling
  - Goal: build an approximate calling context tree (CCT)
  - Expensive to collect via instrumentation



Dynamic call tree

Approximate CCT

Software profiling systems

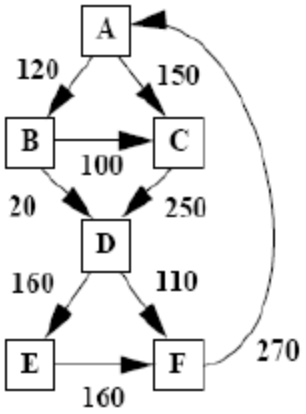
- Sampling-based approach
  - Builds a partial call-context tree (PCCT)
  - Walks the stack on each interrupt up to k frames
  - Overhead 2-4%, Precision more than 90%
  - Disadv:
    - Precision is the correlation of PCCTs for different runs of benchmark (not accurate)
    - Relies on O.S. timer interrupt (Low frequency, slower on faster architectures)

Software profiling systems

- Approximating CCT
  - Uses event-based sampling (method counter)
  - No bound on stack depth sampled
  - High accuracy, O.H. not analyzed (expected to be high)
- Timer/event-based sampling
  - Stride-enabled (mix of timer-based and event-based sampling), O.H. < 0.3%, Accuracy ~ L 60%
- Probabilistic calling context (PCC)
  - Usage: Residual testing, debugging, anomaly detection
  - Instrumentation-based (Sampling is not suitable)
  - No CCT is maintained => O.H. 3%

Path profiling

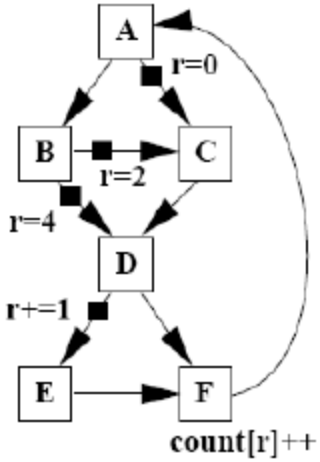
- Frequency of execution paths in CFG
- Exponential in the number of CFG nodes
- Hard to collect by sampling
- Path profile is NOT edge profile



Path	Prof1	Prof2
ACDF	90	110
ACDEF	60	40
ABCDF	0	0
ABCDEF	100	100
ABDF	20	0
ABDEF	0	20

Path profiling

- Efficient path profiling
  - Compact enumeration of path (Path IDs)
  - O.H. average 31%, maximum 97%
  - Disadv: High overhead, suitable only for offline setting



Path	Encoding
ACDF	0
ACDEF	1
ABCDF	2
ABCDEF	3
ABDF	4
ABDEF	5

Path profiling

Online hot path prediction scheme

- Only path head is instrumented instead of full path
- Next Executing Trail (NET) predicted as hot
- Hit rate average: 97% when 10% of execution profiled
- Disadv: Cannot distinguish hot from warm paths (false positives)

Path profiling

- Selective path profiling (SPP)
  - Instrument only paths of interest
  - Usage: Residual testing, profiling different subsets over multiple copies of deployed software
  - Disadv: the set of path is not totally arbitrary
- Preferential path profiling
  - Similar to SPP but paths can be specified arbitrarily
  - O.H. average 15%

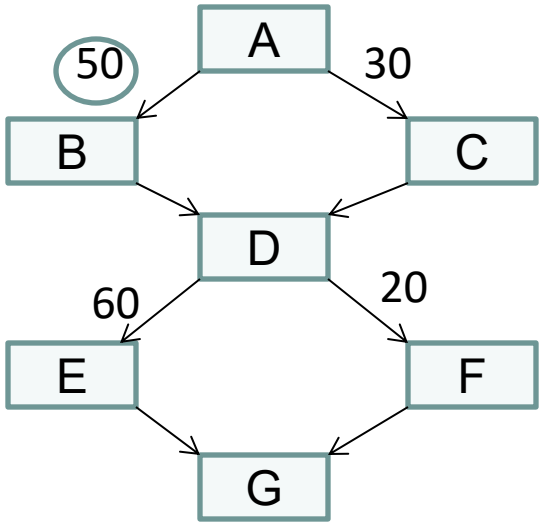
Path profiling

Variational path profiling

- Offline profiling scheme
- Collects execution time variability for paths
- Paths with high variability are good candidates for optimization
- O.H. average: 5%, speedup via simple optimizations: 8.5%
- Disadv: Doesn't report constantly slow paths.

Path profiling

- From edge profile we can find
  - total flow (frequency)
  - upper bound on the flow of each paths
  - lower bound on the flow of each path (definite flow)



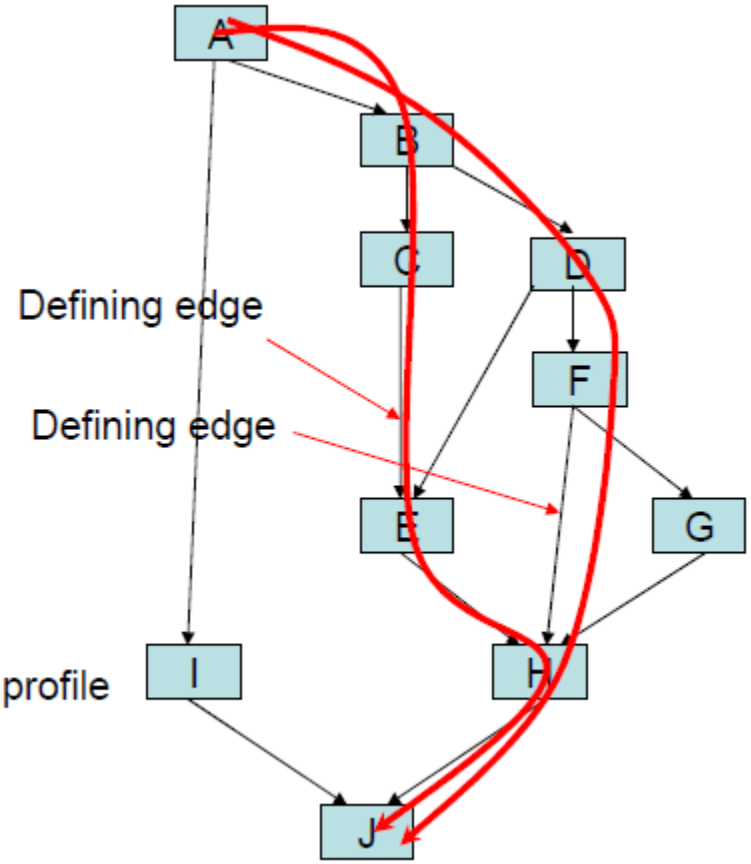
Consider ABDEG:

Max flow = 50

Let ACDFG = 0, ACDEG = 30,
ABDFG = 20
-> Min flow = 30 (Definite Flow)

Path profiling

Defining Edge	Obvious path
AI	AIJ
CE	ABCEHJ
DE	ABDEHJ
FH	ABDFHJ
FG	ABDFGHJ



Obvious path:
Path whose flow can be driven from edge profile

Path profiling

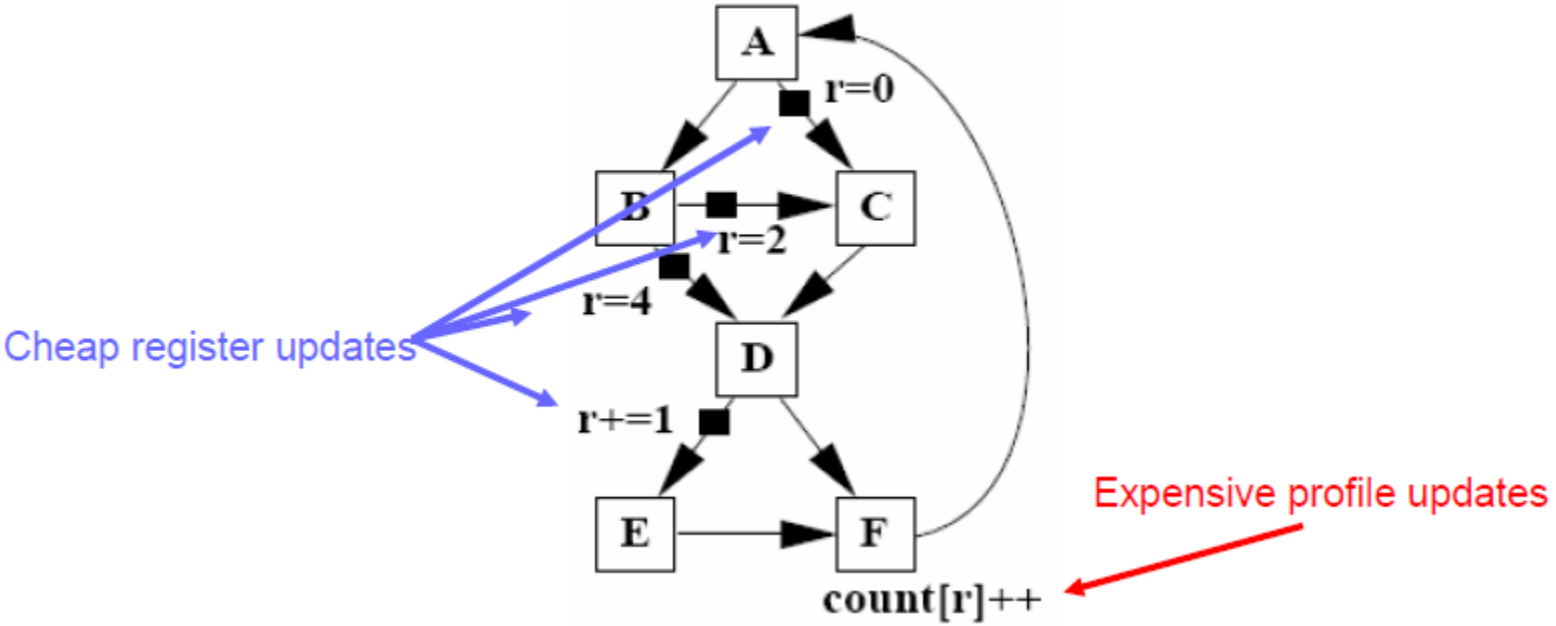
- Targeted path profiling (TPP)
  - Suitable for staged dynamic optimizers
  - Relies on edge profile to:
- Remove obvious paths
- Remove cold edges
  - O.H. 14%, Acc > 97%
  - Disadv: Compilation O.H. 74%

Path profiling

- Practical path profiling (PPP)
  - Reduces amount of instrumentation than TPP
  - Reduce instrumentation overhead
    - Smart path numbering (avoid instrumenting hot edges)
  - O.H. average 5%, accuracy average 96%
  - Disadv: Compilation overhead not reported (expected to be high)

Path profiling

- Two types of instrumentation:



Path profiling

- Path and edge profiling (PEP) [Bond'05]
  - Orthogonal approach that samples profile updates
  - Piggybacks on JikesRVM sampling profiler
  - Edge profile guides instrumentation placing
  - O.H. average 1.2%, 94% accuracy
  - Disadv: VM-specific

Conclusion

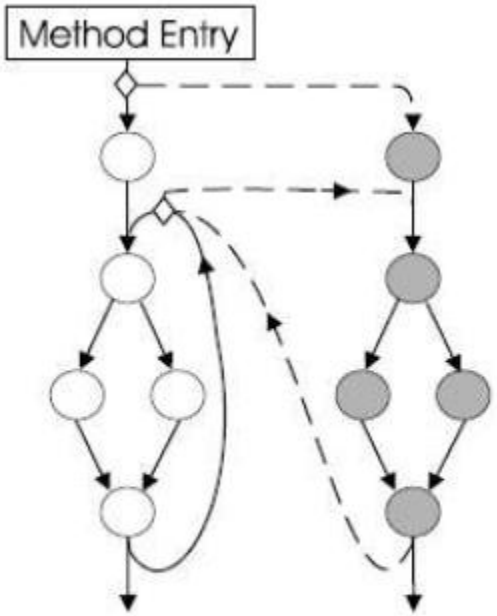
- Programs behavior are hard to understand statically
- Dynamic analysis based on runtime data is needed
- Performance profiling investigates runtime behavior
- Profiling techniques range from exhaustive instrumentation to sampling
- Implementation can be in hardware, software or hybrid
- Context and path profiling techniques

Reducing profiling cost

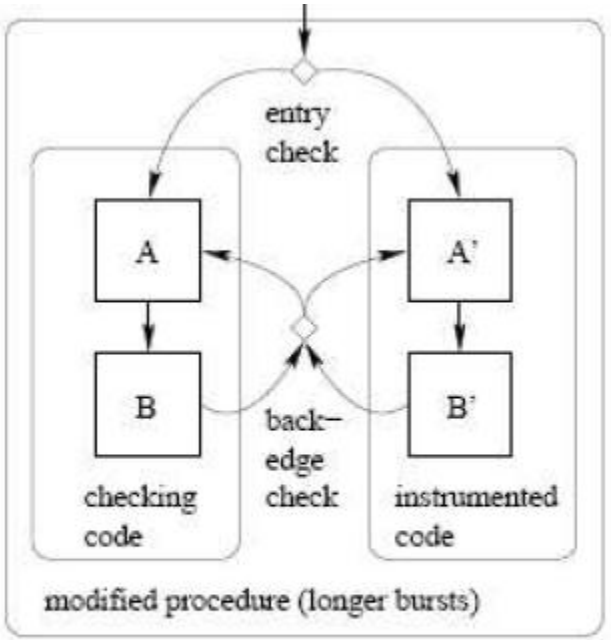
Ephemeral instrumentation

- Temporary instrumentation
  - Shadow profiling
- O.H. < 1%, Accuracy 94% (value-profiling)
  - Profile over adaptive ranges [Mysore ’06]
- Adaptively reducing profile storage overhead

Reducing profiling cost



Source: Arnold '01

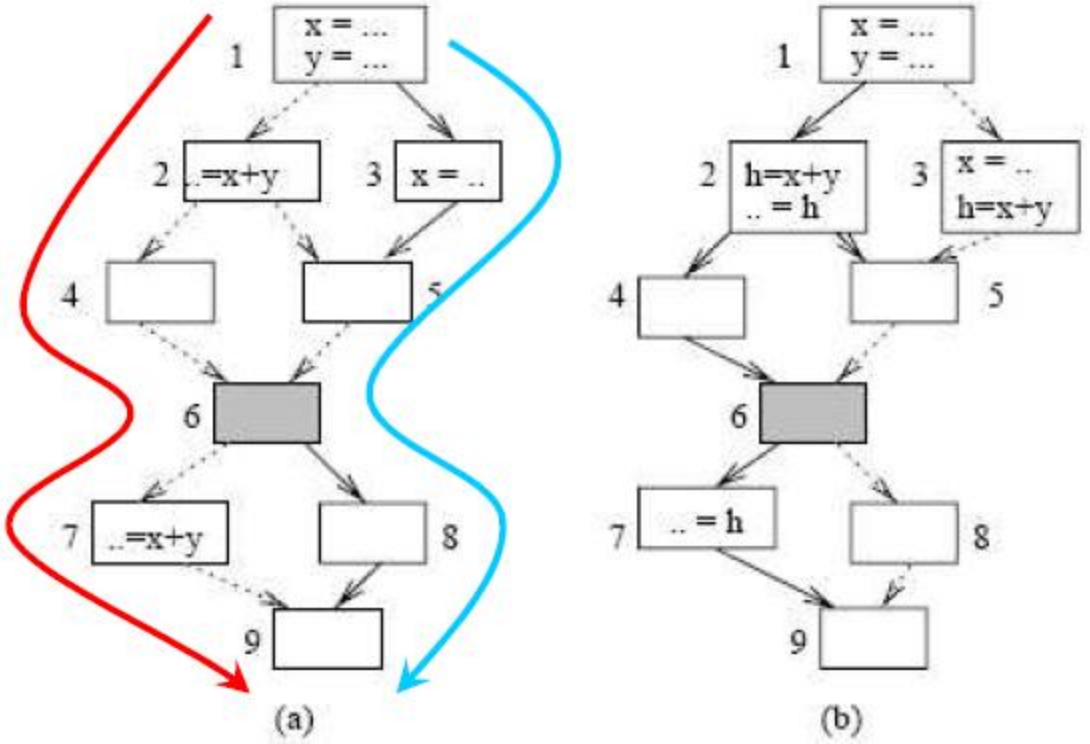


Source: Hirzel '01

Usage examples

- Value profiling and optimization
- Code coverage testing
- Bug isolation
- Understand OO applications
- Offline/Online feed-back directed optimization

Path profiling usage example



Sampling: Triggering mechanism

- Timer-based
  - Easy to implement, Relies on O.S. timer interrupt
    (e.g. 4 ms on Linux 2.6 = 250 samples/sec)
  - Disadv.: Low sampling frequency, independent of processor speed, cannot always correlate with program events

- Event-based
  - Relies on event counting in SW or HW (e.g. HPMs)
  - Correlates with program events
  - Adapts to processor speed

Sampling: Collection mechanism

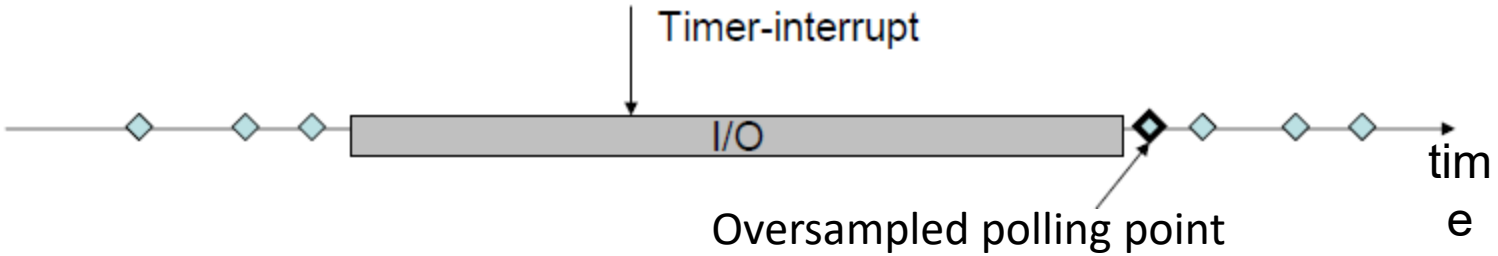
- Polling-based
  - Samples are taken only at polling points
    (e.g. method entries, back-edges, … etc.)

- Disadv.:
  - Not timely
  - Overhead: flag checking
  - Limited accuracy: biased sampling (e.g. polling points after long I/O operations)

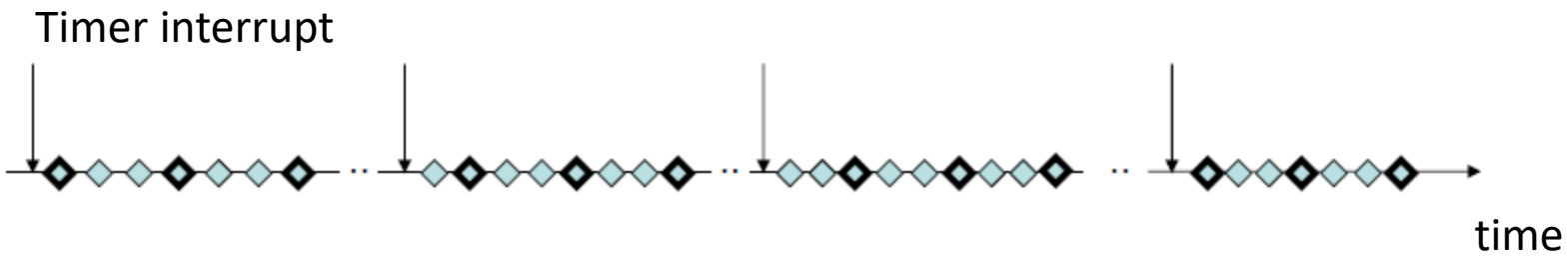
- Immediate
  - Sample is taken right after interrupt is raised

Sampling: Collection mechanism(2)

•Problem with polling sampling



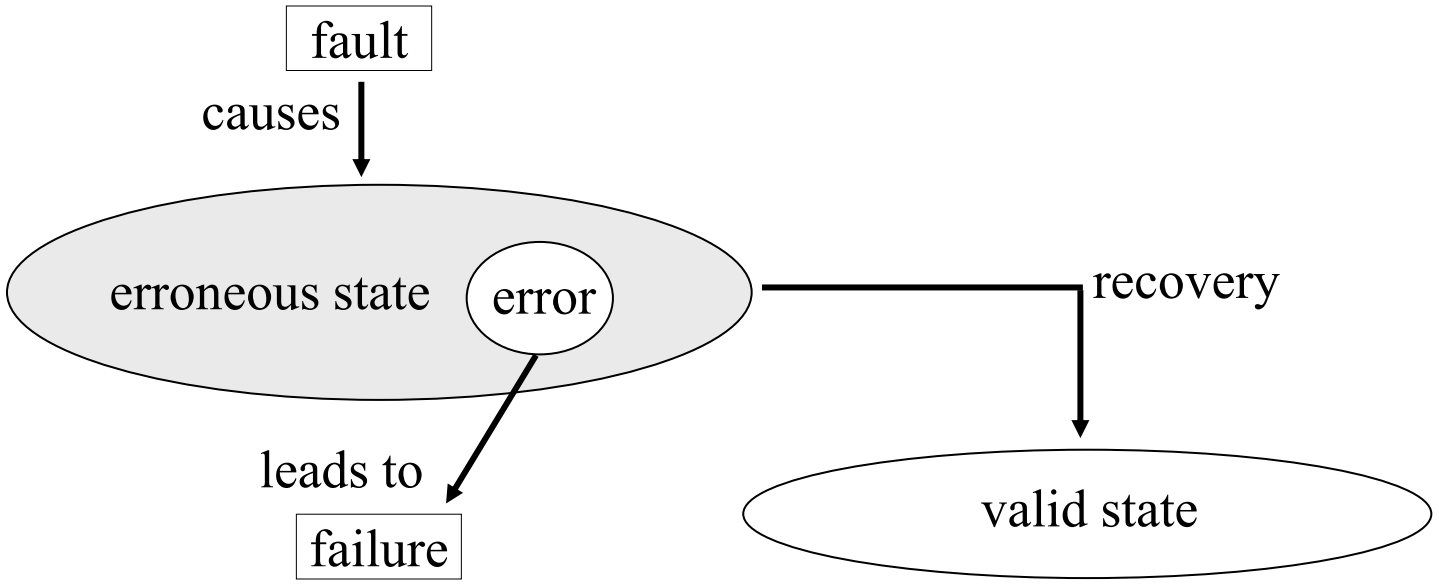
•Arnold Groove Sampling [Arnold '05]



Sample Burst = 3, Stride = 3

Checkpointing-Recovery

Fault Tolerance

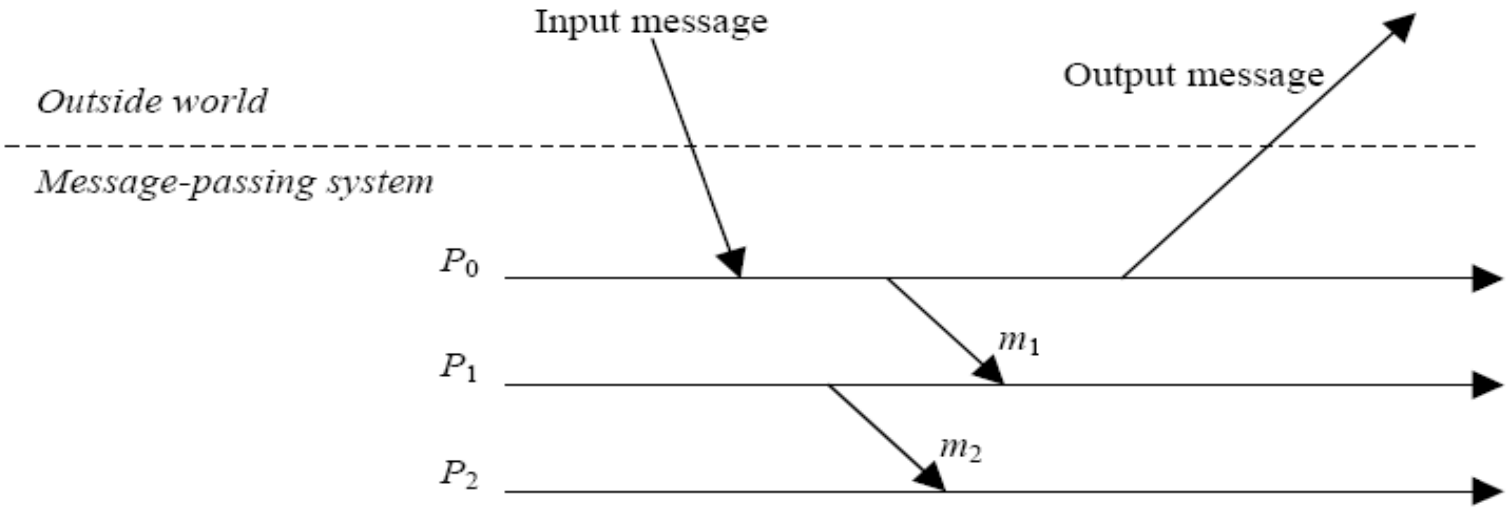


An error is a manifestation of a fault that can lead to a failure.

Failure Recovery:

- backward recovery
  - operationbased (doundoredo logs)
  - statebased (checkpointing/logging)
- forward recovery

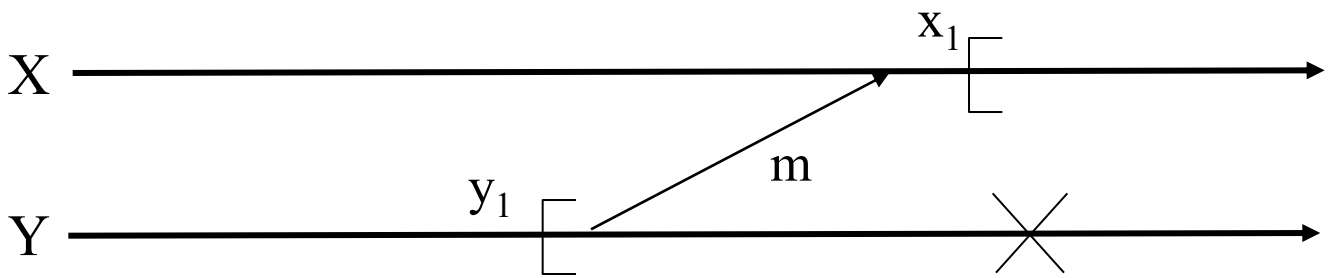
System Model

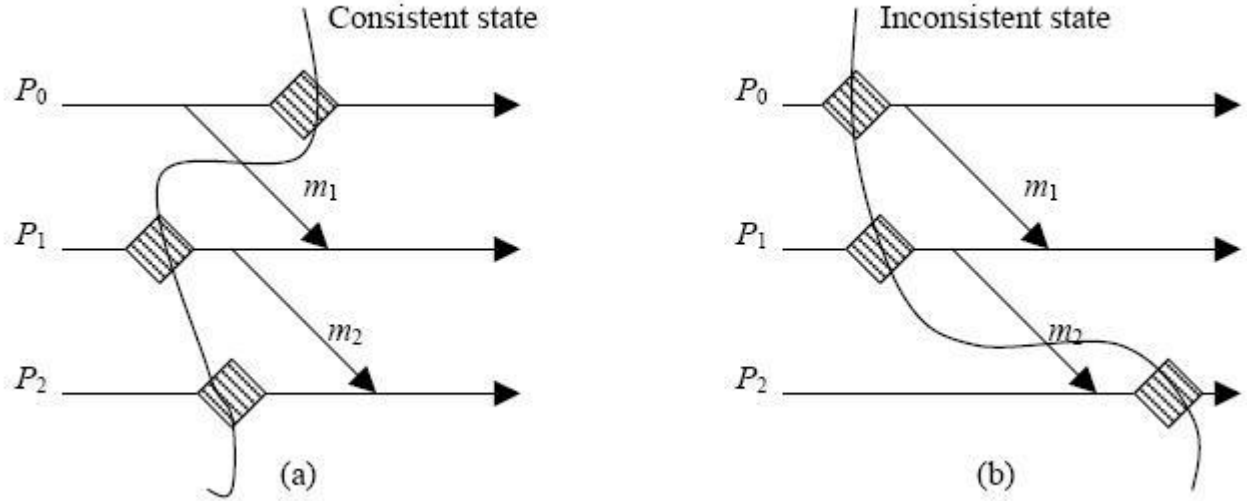


Basic approaches

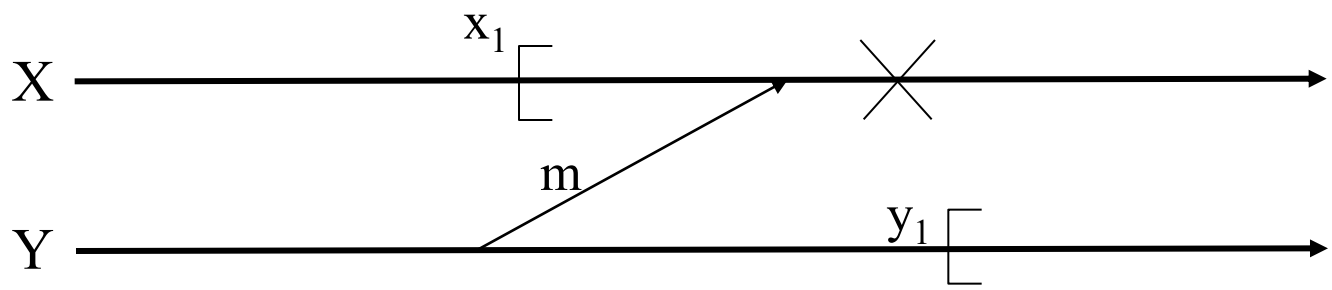
- checkpointing : copying/restoring the state of a process
- logging : recording/replaying messages

Orphan Message





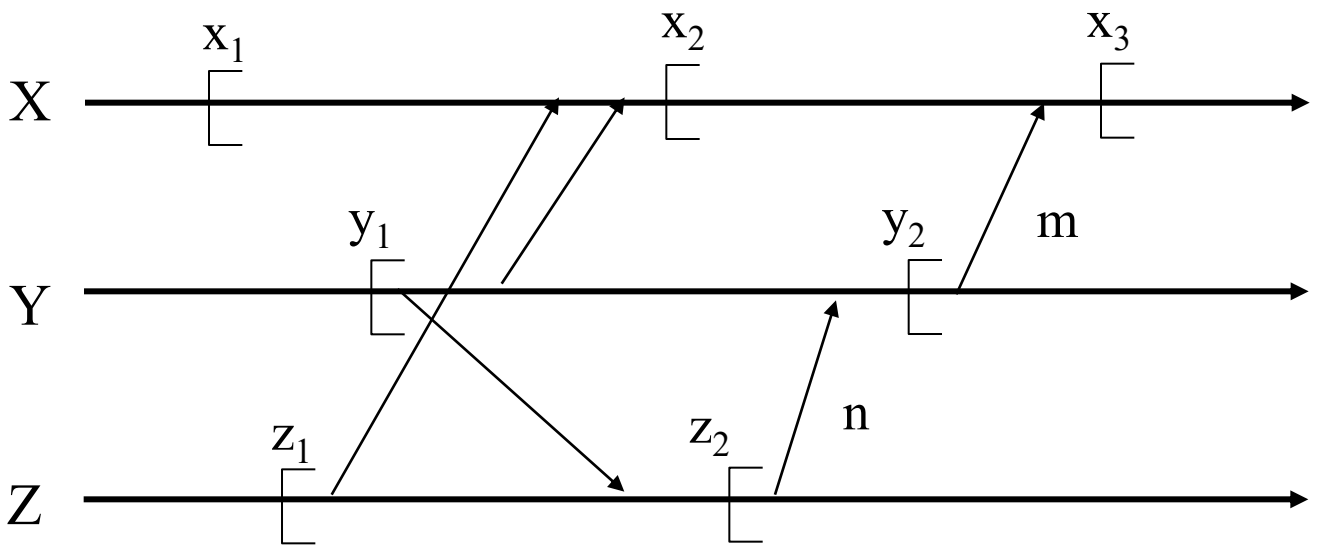
Lost Messages



Regenerating lost messages on recovery:

- if implemented on unreliable communication channels, the application is responsible
- if impelmented on reliable communication channels, the recovery algorithm is responsible

Domino Effect



Cases:

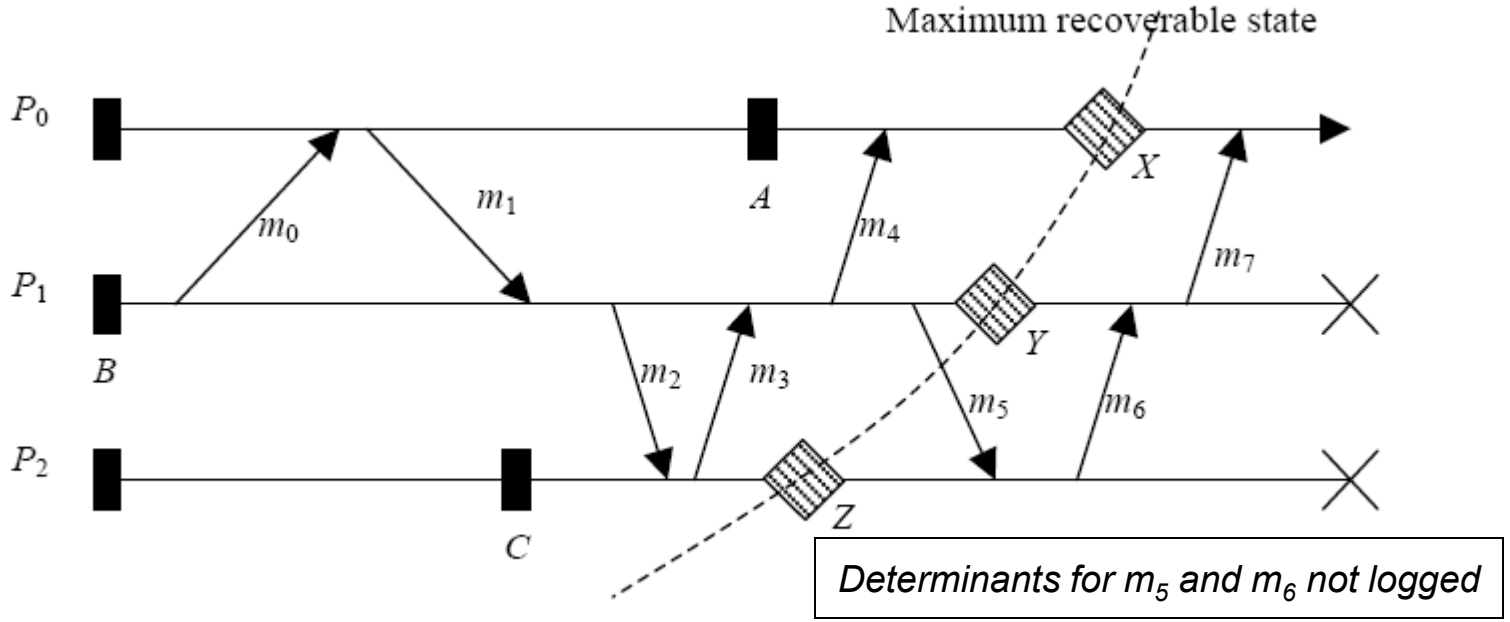
- X fails after x_3
- Y fails after sending message m
- Z fails after sending message n

Other Issues

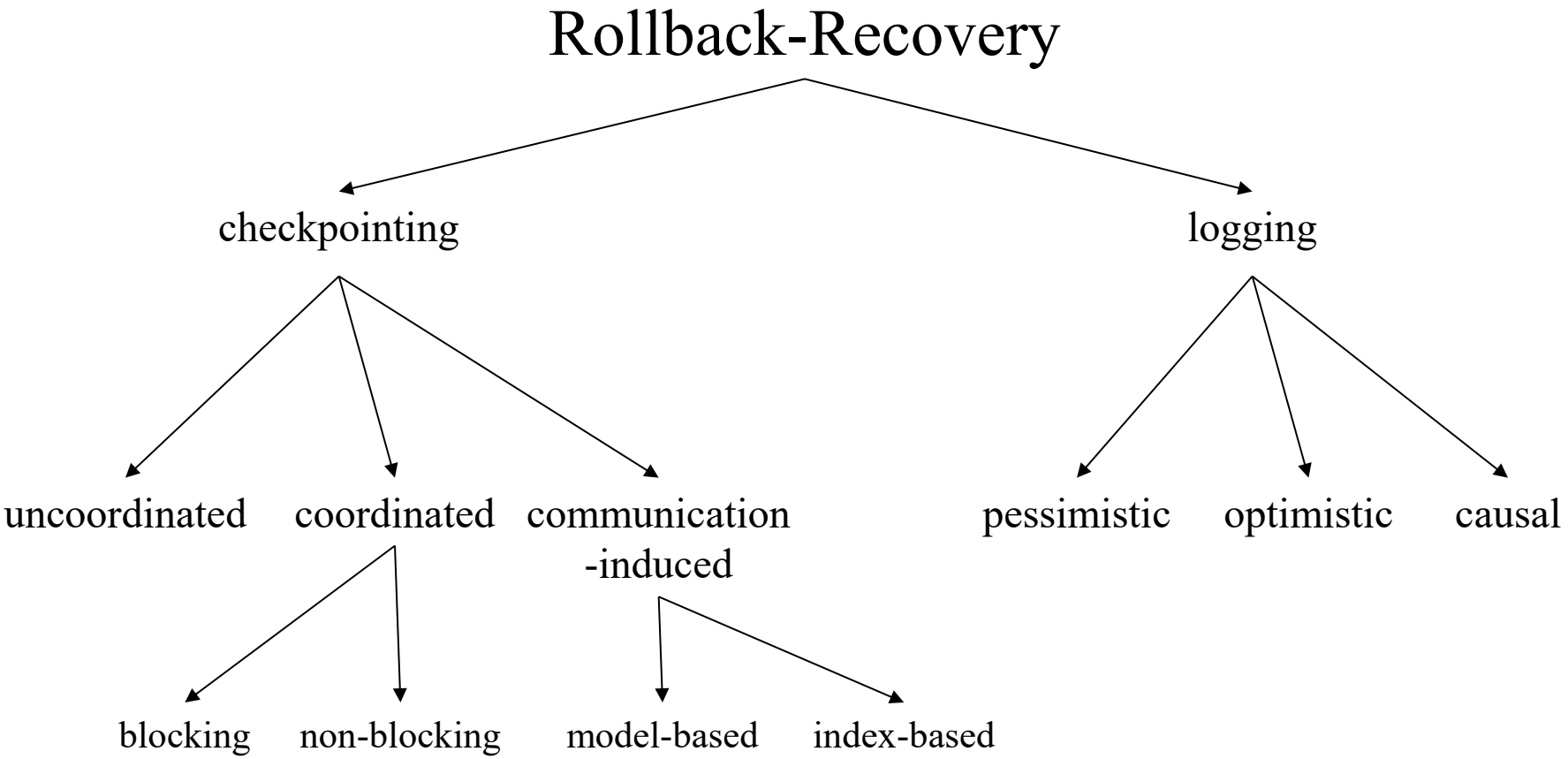
- Output commit
  - the state from which messages are sent to the “outside world” can be recovered
  - affects latency of message delivery to “outside world” and overhead of checkpoint/logging
- Stable storage
  - survives process failures
  - contains checkpoint/logging information
- Garbage collection
  - removal of checkpoints/logs no longer needed

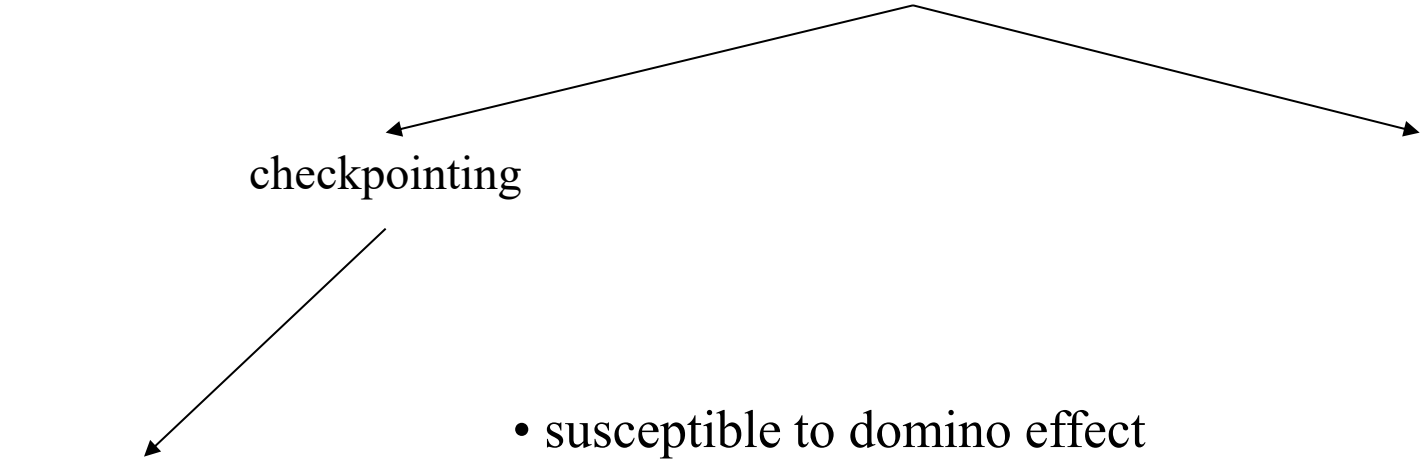
Elements

- Piecewise deterministic (PWD) assumption – the system state can be recovered by replaying message receptions
- Determinant – record of information needed to recover receipt of message



Taxonomy


Rollback-Recovery
checkpointing
logging
uncoordinated
coordinated
communication
-induced
pessimistic
optimistic
causal
blocking
non-blocking
model-based
index-based



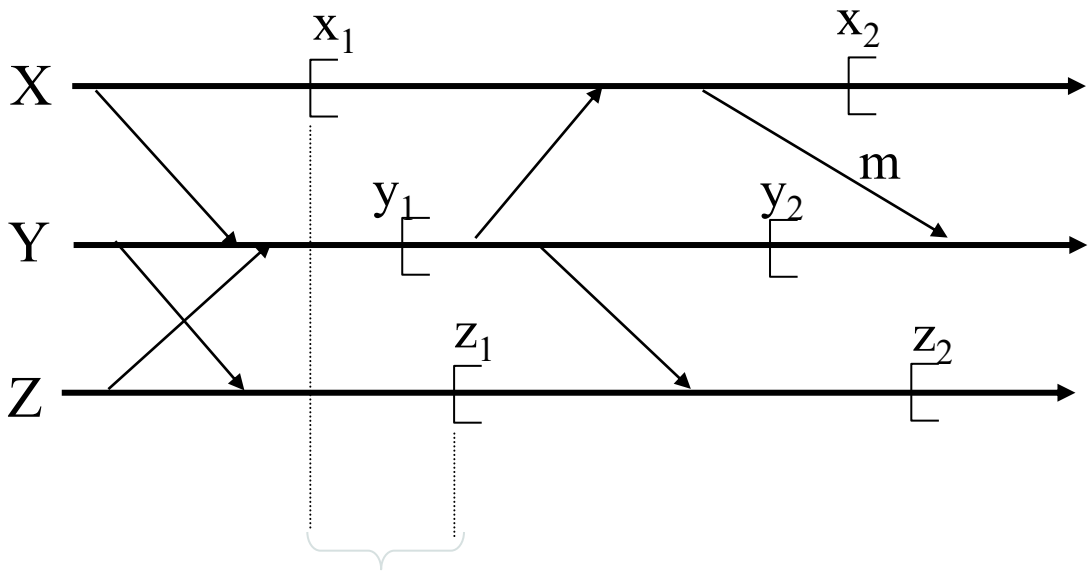
- susceptible to domino effect
- can generate useless checkpoints
- complicates storage/GC
- not suitable for frequent output commits

Rollback-Recovery

checkpointing

coordinated

blocking

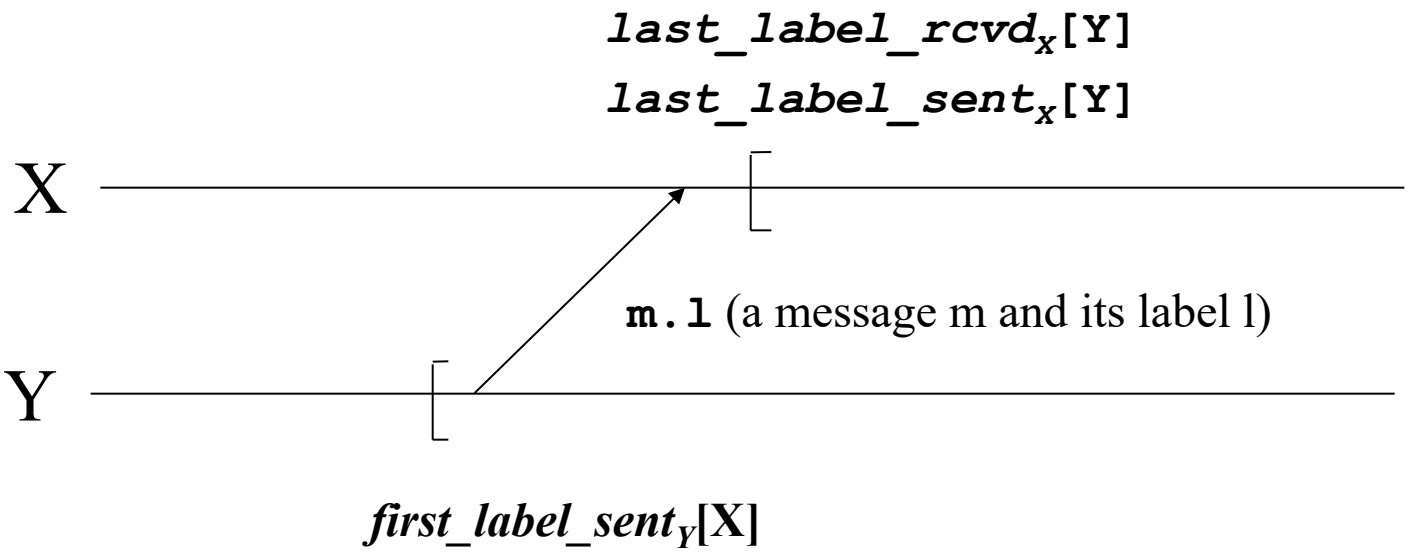


- no messages can be in transit during checkpointing
- $\{x_1, y_1, z_1\}$ forms "recovery line"

Coordinated/Blocking Notation

Each node maintains:

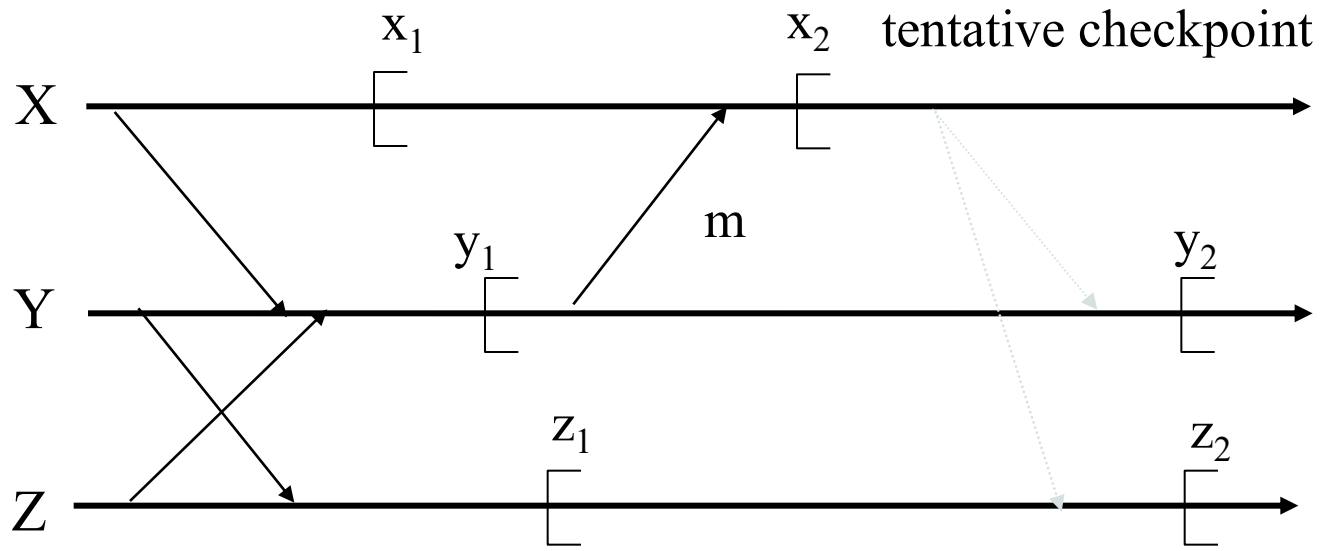
- a monotonically increasing counter with which each message from that node is labeled.
- records of the last message from/to and the first message to all other nodes.



Note: "sl" denotes a "smallest label" that is < any other label and
"ll" denotes a "largest label" that is > any other label

Coordinated/Blocking Algorithm

(1) When must I take a checkpoint?
(2) Who else has to take a checkpoint when I do?



(1) When I (Y) have sent a message to the checkpointing process, X, since my last checkpoint:

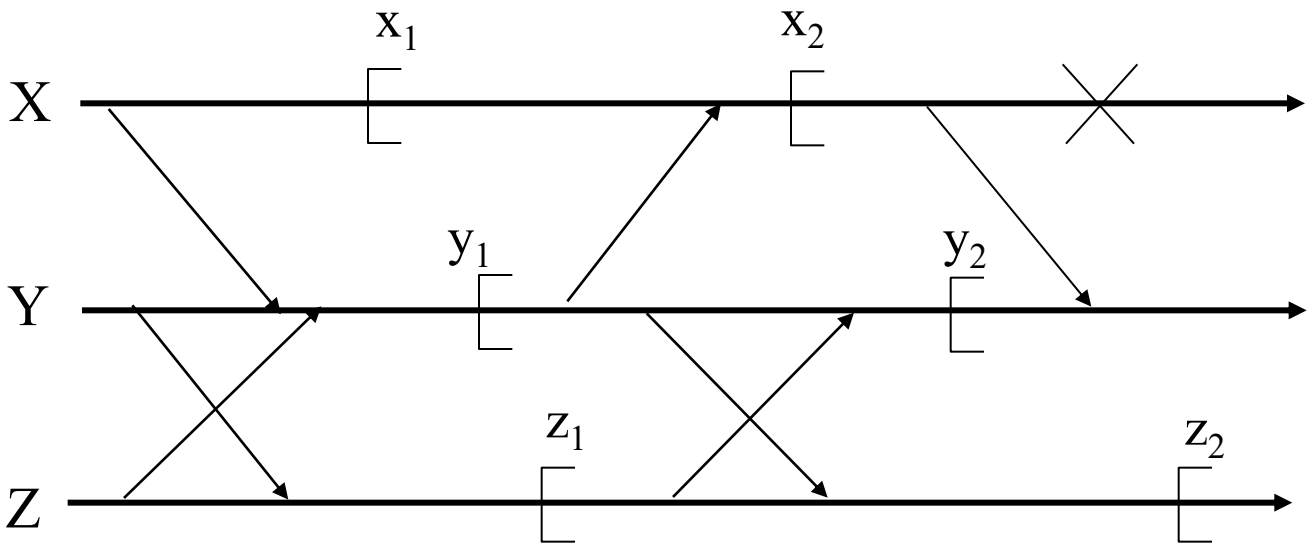
$$last\_label\_rcvd_X[Y] >= first\_label\_sent_Y[X] > sl$$

(2) Any other process from whom I have received messages since my last checkpoint.

$$ckpt\_cohort_X = \{Y \mid last\_label\_rcvd_X[Y] > sl\}$$

Coordinated/Blocking Algorithm

(1) When must I rollback?
(2) Who else might have to rollback when I do?

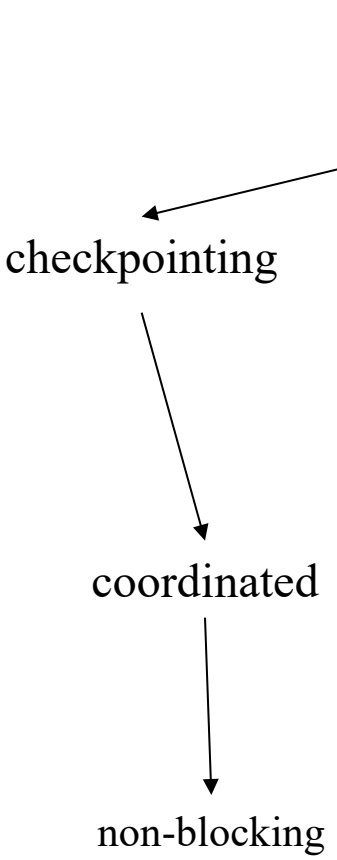


(1) When I ,Y, have received a message from the restarting process,X, since X's last checkpoint.

$$last\_label\_rcvd_Y(X) > last\_label\_sent_X(Y)$$

(2) Any other process to whom I can send messages.

$$roll\_cohort_Y = \{Z \mid Y \text{ can send message to } Z\}$$



Approach:

"tag" message to trigger checkpointing

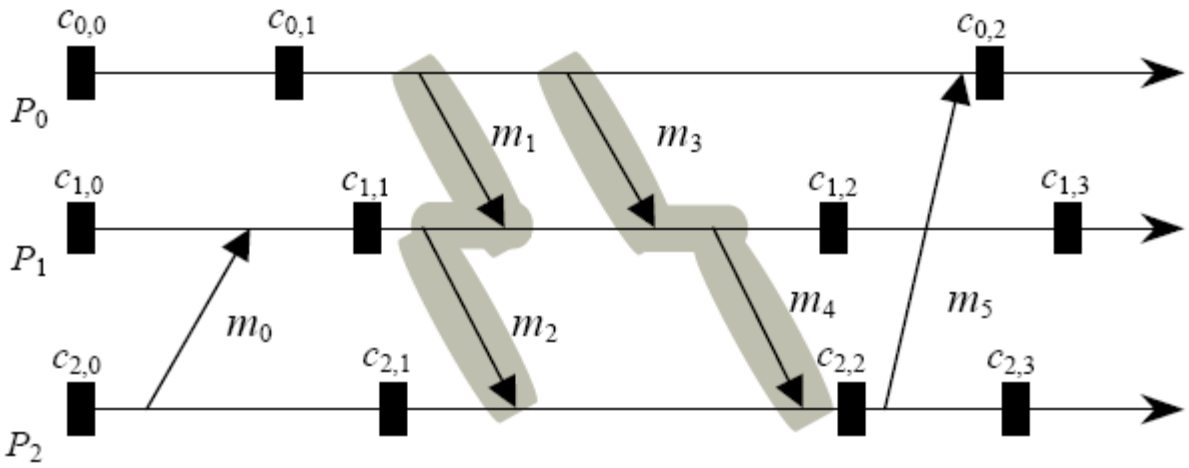
Example:

global-state recording algorithm

Rollback-Recovery

checkpointing

communication
-induced



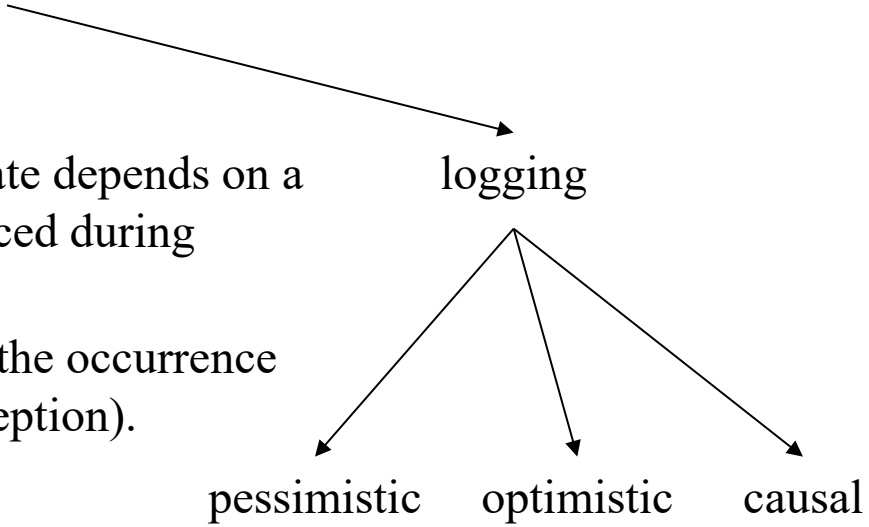
Z-path:[m_1,m_2] and [m_3,m_4]

Z-cycle: [m_3,m_4,m_5]

Checkpoints (like $c_{2,2}$) in a z-cycle are useless

Cause checkpoints to be taken to avoid z-cycles

Rollback-Recovery



Orphan process: a non-failed process whose state depends on a non-deterministic event that cannot be reproduced during recovery.

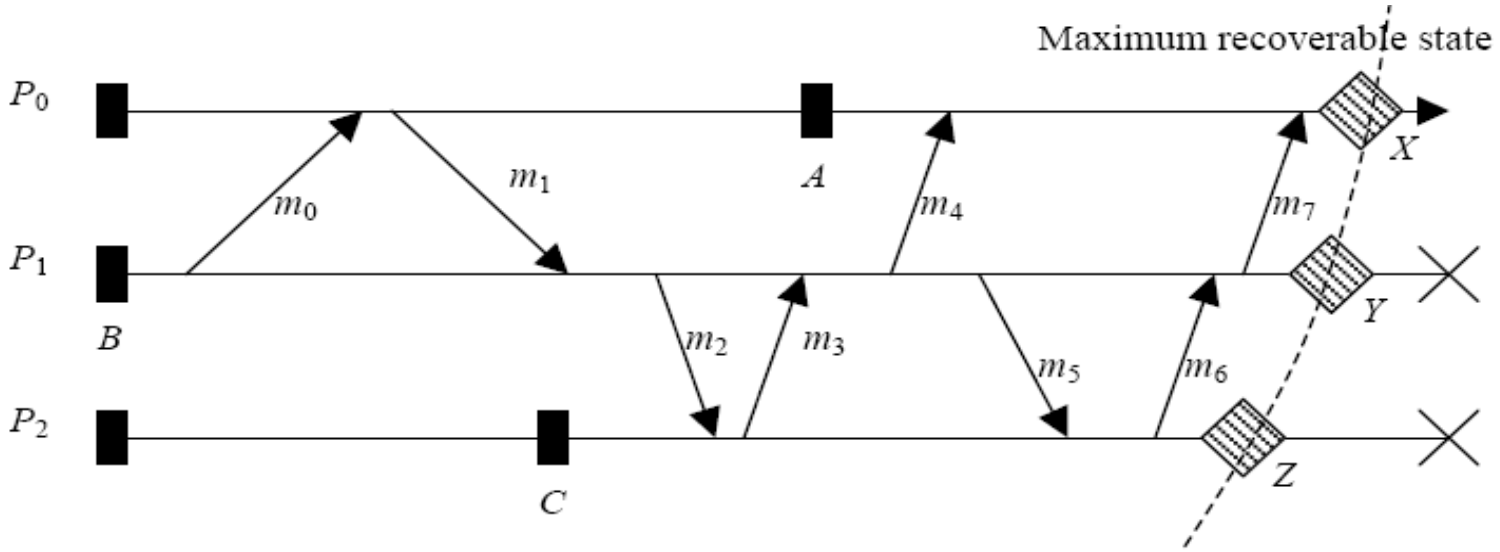
Determinant: the information need to "replay" the occurrence of a non-deterministic event (e.g., message reception).

Avoid orphan processes by guaranteeing:

For all e : not *Stable(e)* => *Depend(e)* < *Log(e)*

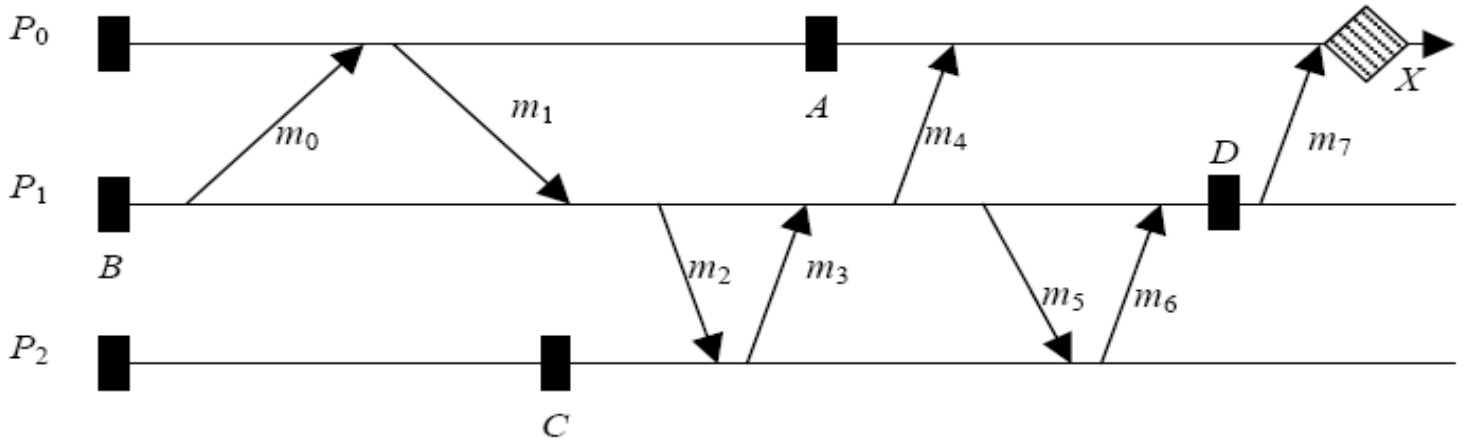
where: *Depend(e)* – set of processes affected by event *e*
Log(e) – set of processes with *e* logged on volatile memory
Stable(e) – set of processes with *e* logged on stable storage

Pessimistic Logging



- Determinant is logged to stable storage before message is delivered
- Disadvantage: performance penalty for synchronous logging
- Advantages:
  - immediate output commit
  - restart from most recent checkpoint
  - recovery limited to failed process(es)
  - simple garbage collection

Optimistic Logging



- determinants are logged asynchronously to stable storage
- consider: P_2 fails before m_5 is logged
- advantage: better performance in failure-free execution
- disadvantages:
  - coordination required on output commit
  - more complex garbage collection

Causal logging

- combines advantages of optimistic and pessimistic logging
- based on the set of events that causally precede the state of a process
- guarantees determinants of all causally preceding events are logged to stable storage or are available locally at non-failed process
- non-failed process “guides” recovery of failed processes
- piggybacks on each message information about causally preceding messages
- reduce cost of piggybacked information by send only difference between current information and information on last message

Program slicing–basic idea…

- *Method used for abstracting from programs.*
  - *Start with a subset of a program's behavior and reduce it to a minimal form that still produces that behavior.*
- *The reduced program is called a "slice".*
- *Problem – finding a slice is in general unsolvable.*

Automatic slicing

- *Specifying what behavior is of interest to us…*
  - *Usually expressed as the values of some variables at some statement in the program…called a* “slicing criterion”.
- *To find what all code could influence a variable in a statement,* ***data flow analysis*** *is extremely useful!*

Slices & slicing criteria

- *Slicing criterion – a pair <i, v>*
  - *"i" is the number of the statement & "v" is the set of variables to be observed at statement "i".*
- *Desirable properties of a slice:*
  - *Must have been got from the actual program by deletion.*
  - *Behavior of slice as observed thro' the window of the criterion must correspond to original program behavior!*

Examples of slices

- *A program may have several slices depending on the criterion for slicing…*

Original program:

```
1 begin
2 read(x,y)
3 total := 0.0
4 sum := 0.0
5 if x <= 1
6       then sum := y
7       else begin
8            read(z)
9            total := x*y
10           end
11 write(total, sum)
12 end.
```

Criterion: <12, z>

```
begin
read(x,y)
if x <= 1
     then
     else read(z)
end.
```

Criterion: <9, x>

```
begin
read(x,y)
end.
```

Criterion: <12, total>

```
begin
read(x,y)
total := 0
if x <= 1
     then
     else total := x*y
end.
```

Dataflow

- *Data flow analysis*
  - *To find slices, a program is traced backwards, finding possible influences on variables…similar to reaching definition analysis.*
  - *In this analysis, the following definitions are most important:*
    - *USE(n) – set of variables whose values may be referenced at node (or statement) n.*
    - *DEF(n) – set of variables whose values may be changed at node n.*

Inter-procedural slicing

- *Consists basically of two steps:*
  - *A single slice of the procedure containing the slicing criterion is made.*
  - *Procedure calls from within this procedure are sliced using new criteria.*

Slicing techniques

- *Static*
  - *Here, slices are computed statically using a dependence graph.*
- *Dynamic*
  - *Dynamic data dependence information is traversed to compute the slices…this info is constructed using an **execution trace** of the program.*
  - *Hence, a dynamic slice is a set of statements that **did** affect the value of a variable for **one** specific input.*

Effectiveness of dynamic slicing

Program	Static / Dynamic (25 slices)		
	AVG	MIN	MAX
126.gcc	5448	3.5	27820
099.go	1258	2	4246
134.perl	66	1	1598
130.li	149	1	1436
008.espresso	49	1	1359

Sometimes, both get similar results…

Sometimes, static gets really huge results

On an average, static slices are much larger!

Example

```
Funcion f(N)
1:  z=0
2:  a=0
3:  b=2
4:  p=&b
5:  for i= 1 to N do
6:     if (i%2==0) then
7:        p=&a
       endif
8:     a=a+1
9:     z=2*(*p)
    done
10: print(z)
```

For input N=1,

1_1: z=0	[z=0]
2_1: a=0	[a=0]
3_1: b=2	[b=2]
4_1: p=&b	[p=&b]
5_1: for i = 1 to N do	[i=1]
6_1: if (i %2 == 0) then	[false]
8_1: a=a+1	[a=1]
9_1: z=2*(*p)	[z=4]
10_1: print(z)	[z=4]

Static Slice (<10, z>) = *{1, 2, 3, 4, 7, 8, 9, 10}*

Dynamic Slice (<input = 1, variable = z, execution point = 10_1>)

= *{3, 4, 9, 10}*

Imprecision in algorithms

- *Data dependence analysis is sometimes conservative…this results in imprecise, larger slices.*
- *More precision implies more expensive computation…* ☹

Precision comparison

Program	Algo-II / Precise Algo			Algo-I / Precise Algo		
	AVG	MIN	MAX	AVG	MIN	MAX
126.gcc	419	1	5188	2698	1.75	13759
099.go	39	1	946	1413	1	3328
134.perl	8	1	96	40	1	974
130.li	13	1	42	143	1	1374
008.espresso	4	1	49	368	1	1134
Average	96	1	1264	932	1.15	4113

Imprecise algorithms to precise algorithm comparison …w.r.t. slice sizes

Precise Dynamic Slicing Algorithms

- *This paper presents algorithms that are precise and yet have reasonable computation cost.*

Basic approach…

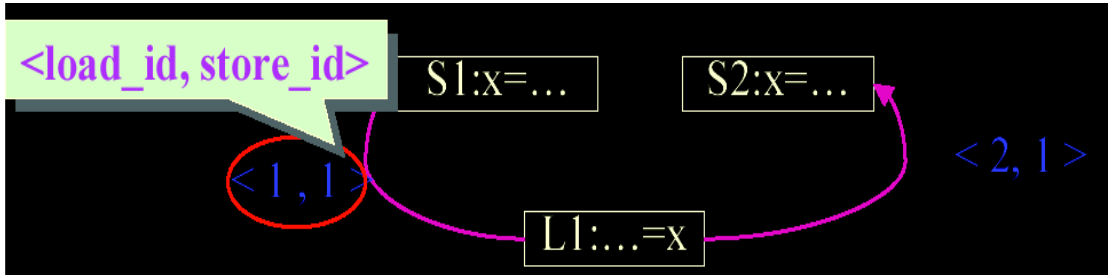
- *Execution trace*
  - *Captures runtime info that can be used for dynamic slicing…control flow trace + memory reference trace.*
- *Slice computation…*
  - *Execution trace is processed to identify dynamic dependences that are exercised during the program execution.*
  - *Slicing request given in terms of a variable & a memory address.*
- *Goals*
  - *Allow for computation of precise dynamic slices.*
  - *Support computation of a slice for any variable/memory address at **any execution point**.*

The three algorithms…

- *Full preprocessing*
  - *Execution trace is completely processed to generate all dependence information.*
- *No preprocessing*
  - *Execution trace is not at all preprocessed. All dependence info is generated dynamically as required.*
- *Limited preprocessing*
  - *Some amount of preprocessing is performed.*

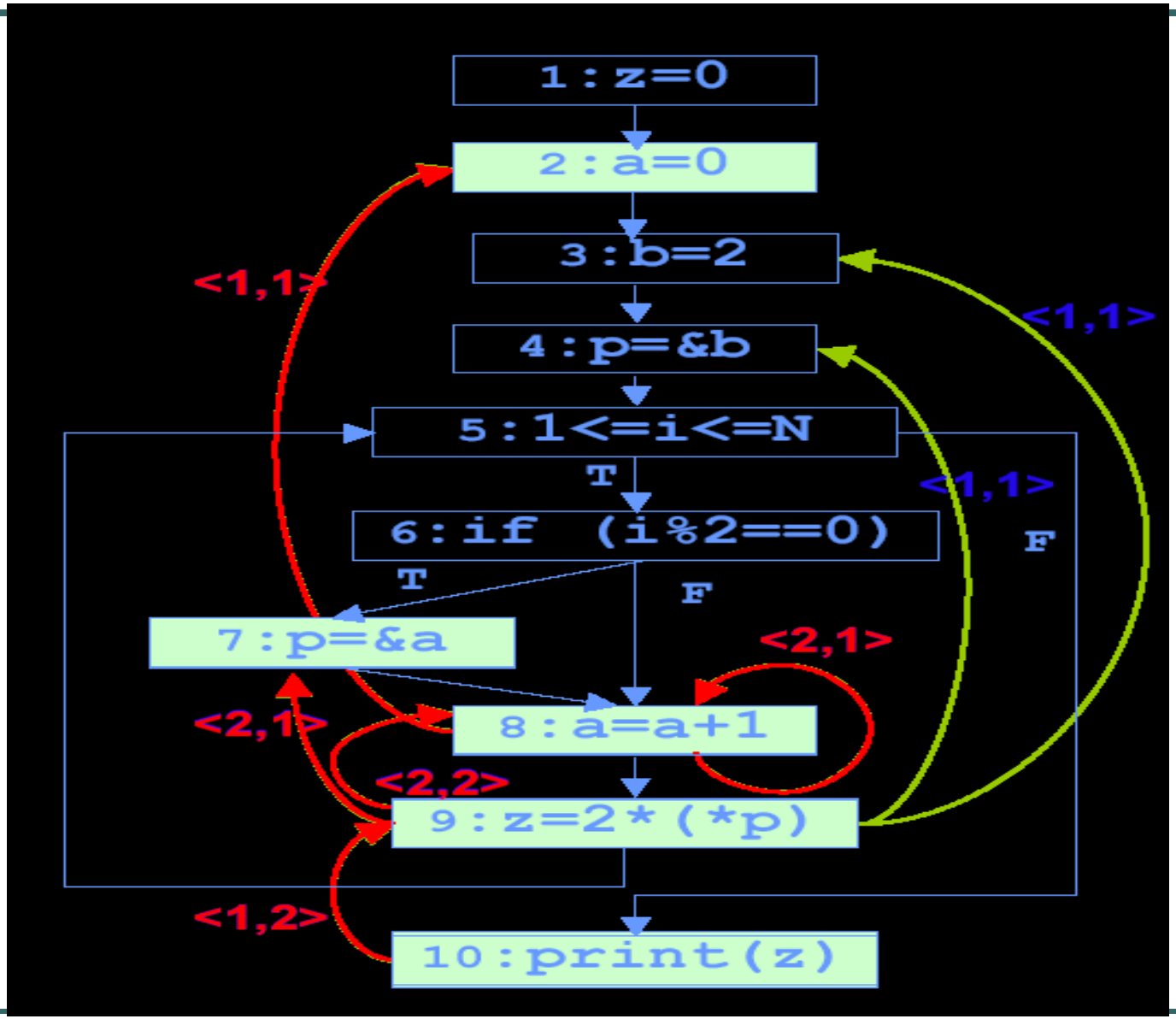
Full preprocessing (FP)

- *Preprocessing*
  - *Execution trace is preprocessed completely… dependence edges in the graph are labeled with instance numbers.*

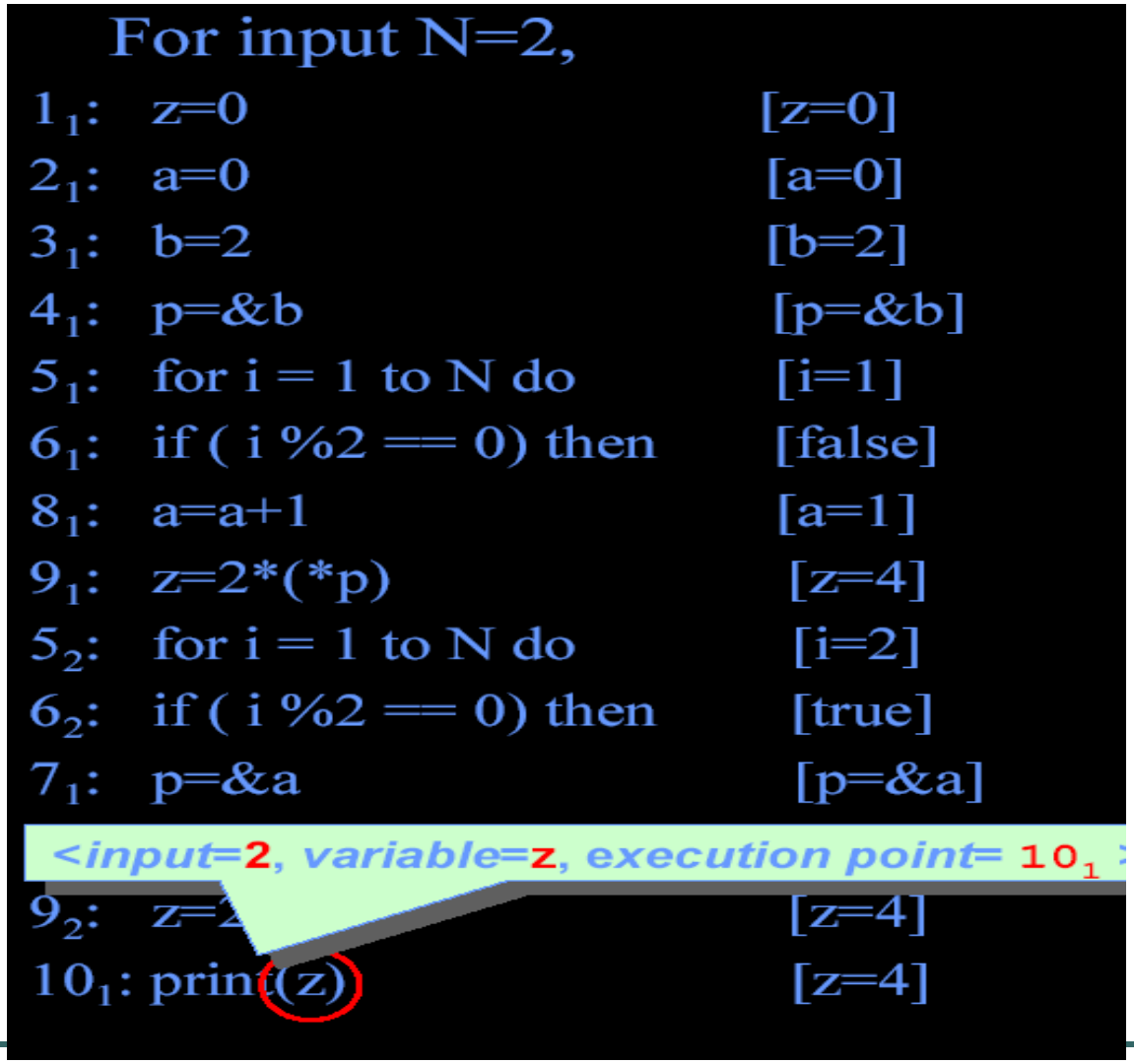


- *Slicing*
  - *Instance labels are used to traverse only relevant edges.*

Example


1 : z=0
2 : a=0
3 : b=2
4 : p=&b
5 : 1<=i<=N
T
6 : if (i%2==0)
T
F
7 : p=&a
8 : a=a+1
9 : z=2*(*p)
10 : print(z)
<1,1>
<1,1>
<1,1>
F
<2,1>
<2,1>
<2,2>
<1,2>

Execution trace…



Characteristics

- *Advantages*
  - *Very easy and fast to compute a dynamic slice at any execution point.*
- *Disadvantages*
  - *Dynamic dependence graph is very large for real programs…so, this scheme is* ***not practical****.*

No preprocessing (NP)

- *Completely demand-driven*
- *Space requirements are very less when compared to FP.*
- *Slower than FP.*
- *Computations may be repetitive at times. Caching may be used to try and avoid this.*

Limited preprocessing (LP)

- *Strikes a balance between preprocessing and slicing costs.*
- *Process:*
  - *Trace is divided into trace blocks of a fixed size.*
  - *Every trace block stores a summary of downward exposed definitions of variables & memory addresses.*
  - *This summary helps reduce slicing time.*

Effectiveness of LP

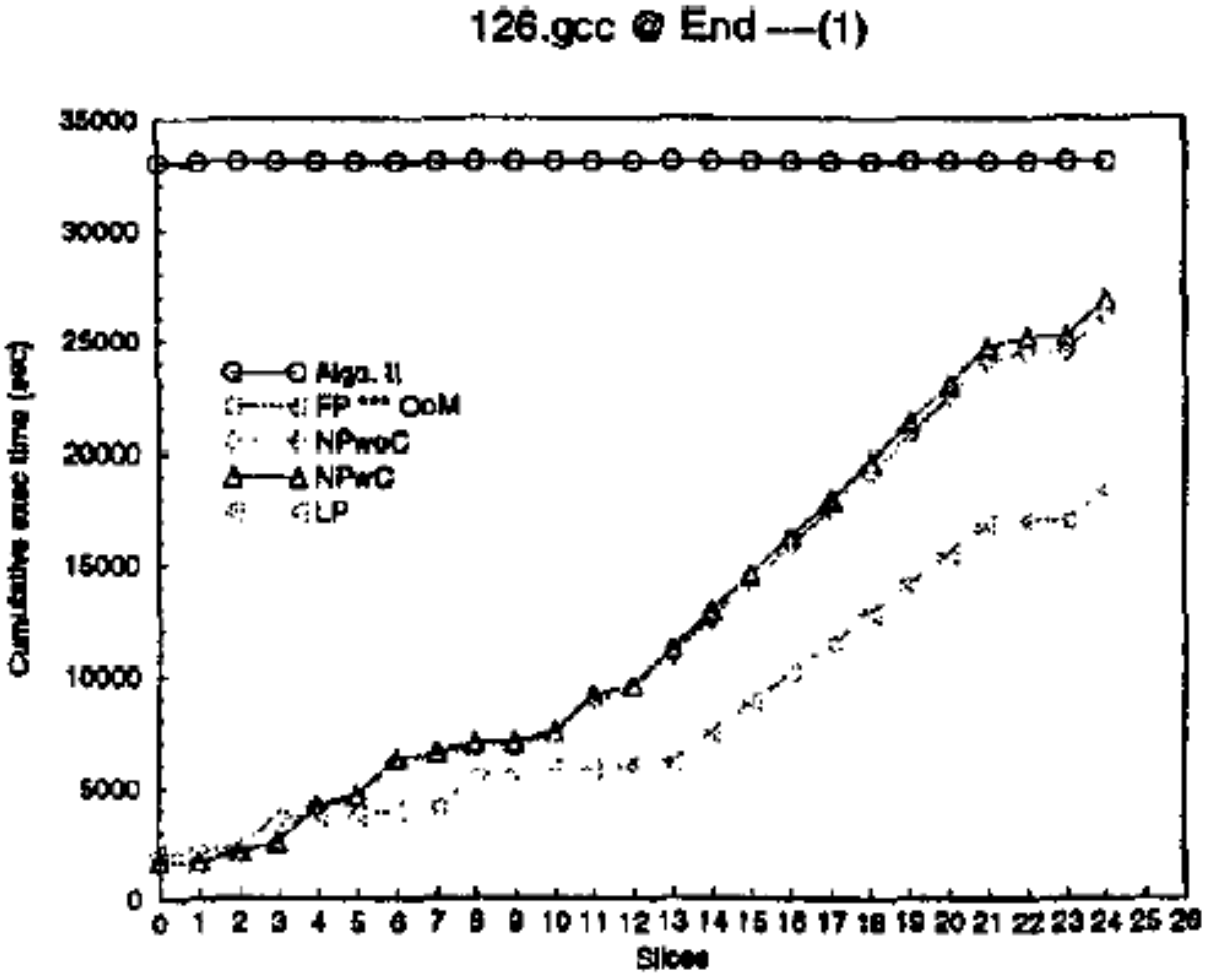
Program`	% Blocks Skipped		
	Input (1)	Input (2)	Input (3)
126.gcc	90.43	97.63	89.39
099.go	57.52	65.38	91.7
134.perl	92.42	98.99	98.26
130.li	99.02	99.51	99.70
008.espresso	96.6	97.15	98.68
Average	87.20	91.73	95.65

Trace blocks skipped by LP

Experimental results

- *Precise slicing algorithms*
  - *Slice sizes*
    - *Number of statements in dynamic slice is much less than actual number of program lines, as expected!*
    - *This is consistent for all inputs considered.*
  - *Slicing time…comparison between FP, NP and LP*
    - *FP usually runs out of memory…never reaches slicing stage.*
    - *For NP, caching sometimes help and sometimes doesn't.*
    - *LP execution times increase much more slowly than NP.*
    - *Preprocessing cost is more than compensated by slicing time reduction.*

Result graphs



Comparison of execution times for each algorithm for a benchmark.

LP vs imprecise algorithm

- *Both solve the slicing problem fully…no memory issues.*
- *Results:*
  - *Slice sizes much smaller for LP algorithm.*
  - *Slice independent comparison shows that the imprecise algorithm has several unnecessary dependences… hence the imprecision!*
  - *Execution times are greater for LP for large slices and smaller than the imprecise algorithm for smaller slices.*

Execution Indexing (EI)

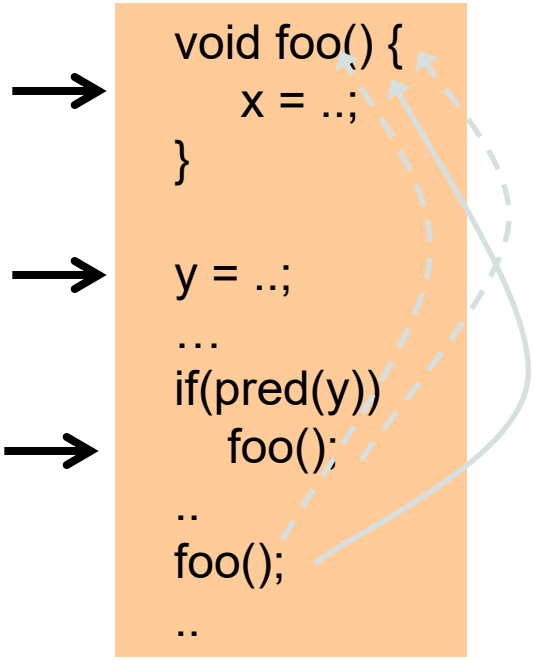
- A technique that aligns two executions so that equivalent points in two executions can be identified.
- Multiple scenarios:
  - Run a program twice on one input (with perturbations);
  - Run two versions of a program on one input;
  - Run a program twice with different inputs;

Why EI – Case I

- Running the same program with the same input

  -- Setting breakpoints in debugging

1. break at the 1st instance
2. restart, “set y=<val>”
3. continue running.
4. stopped, but not at the desired point.



- Setting breakpoints for multiple threads.

```
producer thread {
     while (…) {
       s=generate_task ( …);
       q.enqueue (s);
     }
}

consumer thread {
   while (s=dequeue( )) {
     work_on_task(s);
  }
}
```

Assume there are 1 producer thread and 2 consumer threads, T1 and T2, and 5 tasks – s1, s2, s3, s4, and s5.

In one execution:

T1: s1

T2: s2, s3, s4, s5

In another execution:

T1: s2, s3

T2: s1, s4, s5

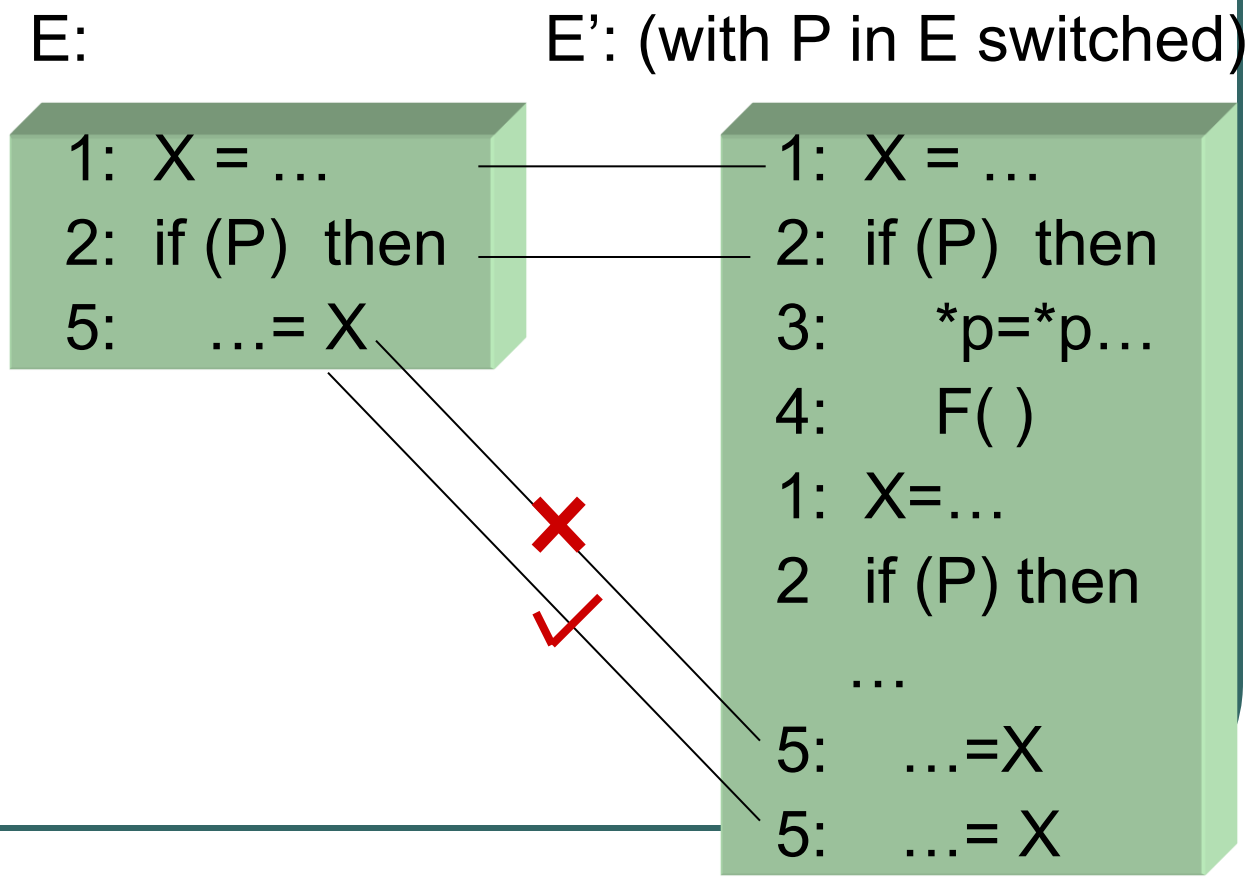
Why EI – Case II & III

- Running two program versions with the same input
  - Program de-obfuscation
  - Debugging compiler
  - Malware detection
  - Debugging regression errors.
- Running a program twice with different inputs
  - Comparison based debugging

Using S_i

- Identify an execution point using a pair
  - <statement, instance>

```
F ( ) {
  1:  X = …
  2:  if (P) {
  3:      *p = *p …
  4:      F( )
  }
  5:  …= X
}
```



Using Calling Context and Instance

- Lets look at a more sophisticated proposal
  - Identify an execution point using a triple
    - <calling context, statement, instance>

```
1. while (…) {
2.    fgets (buf, 256, h);
3.    for (i;i<strlen(buf);i++){
4.        F(buf[i]);
5.    }
6. }
7. F (char c) {
8     …=c;
9. }
```

In one execution, the input file is:

ab\n

1

In the other execution, the input file is:

a\n

1

Lets look at <[(main, 4)], 8, 2>

Basic Idea

- Align executions top-down, region by region
  - **Region**: executed statements between a predicate instance and its immediate post-dominator or a function entry and the corresponding exit form a region
    - A statement instance x_i DCD on the predicate instance leading x_i 's enclosing region.
    - Regions are either nested or disjoint, never overlap.

Basic Idea

- Align executions top-down, region by region
  - At the highest level

```
F ( ) {
  1:  X = …
  2:  if (P) {
  3:       *p = *p …
  4:       F( )
      }
  5:  …= X
}
```

E:

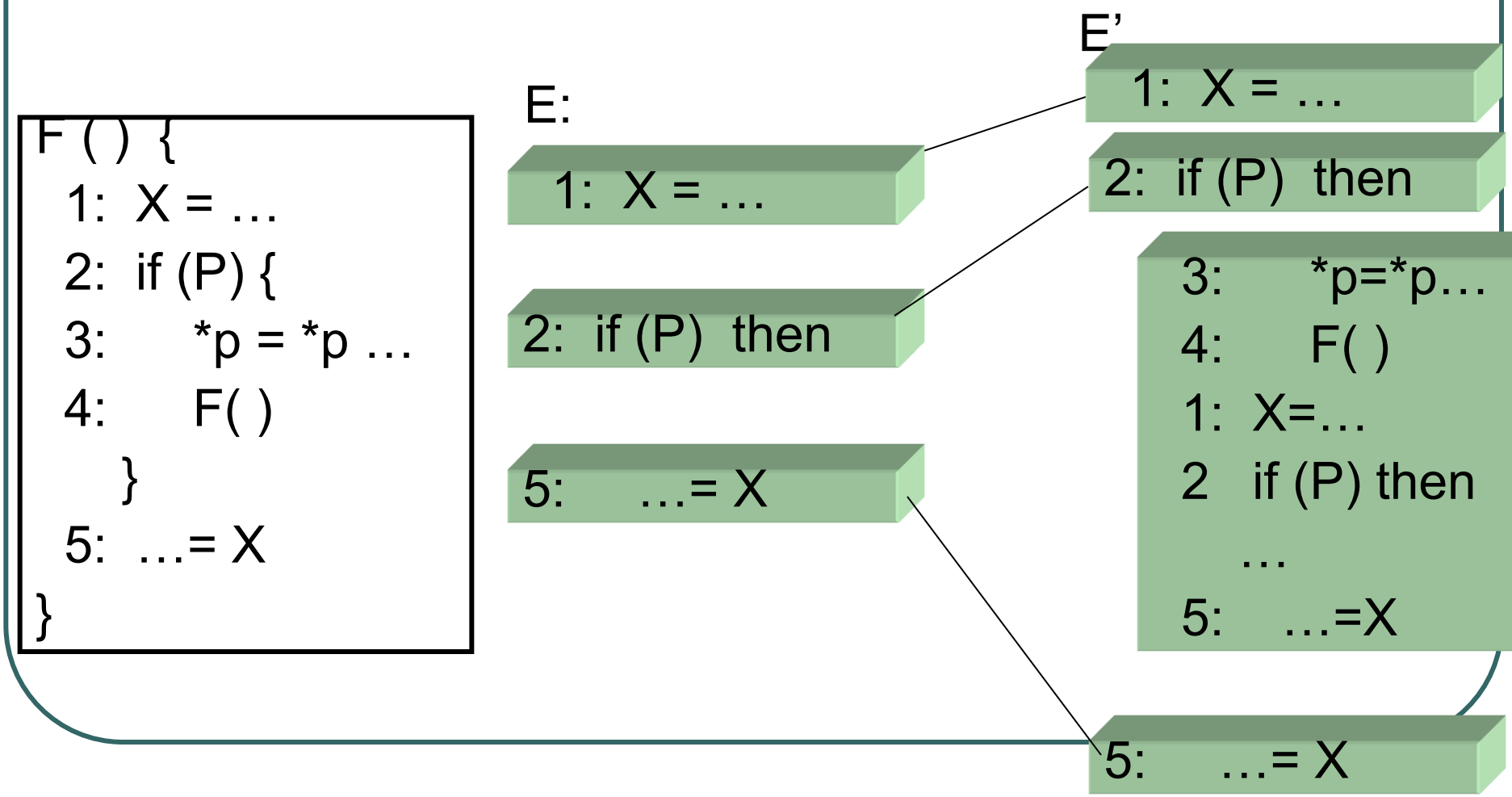
```
1:  X = …
2:  if (P)  then
5:      …= X
```

E’

```
1:  X = …
2:  if (P)  then
3:       *p=*p…
4:       F( )
1:  X=…
2   if (P) then
      …
5:     …=X
5:     …= X
```

Basic Idea

- Align executions top-down, region by region
  - One level down



```
1. while (…) {
2.    fgets (buf, ...);
3.    for (…strlen(buf);
4.        F(buf[i]);
5.    }
6. }
7. F (char c) {
8     …=c;
9. }
```

Input:

ab\n

1

```
while(…)
    fgets(buf);
    for (i…)
        F(buf[i]);
            …=c;
    for (i…)
        F(buf[i])
            …=c;
    for (i…)
while (…)
    fgets (buf)
    for (i…)
        F (buf[i]);
            …=c;
    for (i…)
```

Input:

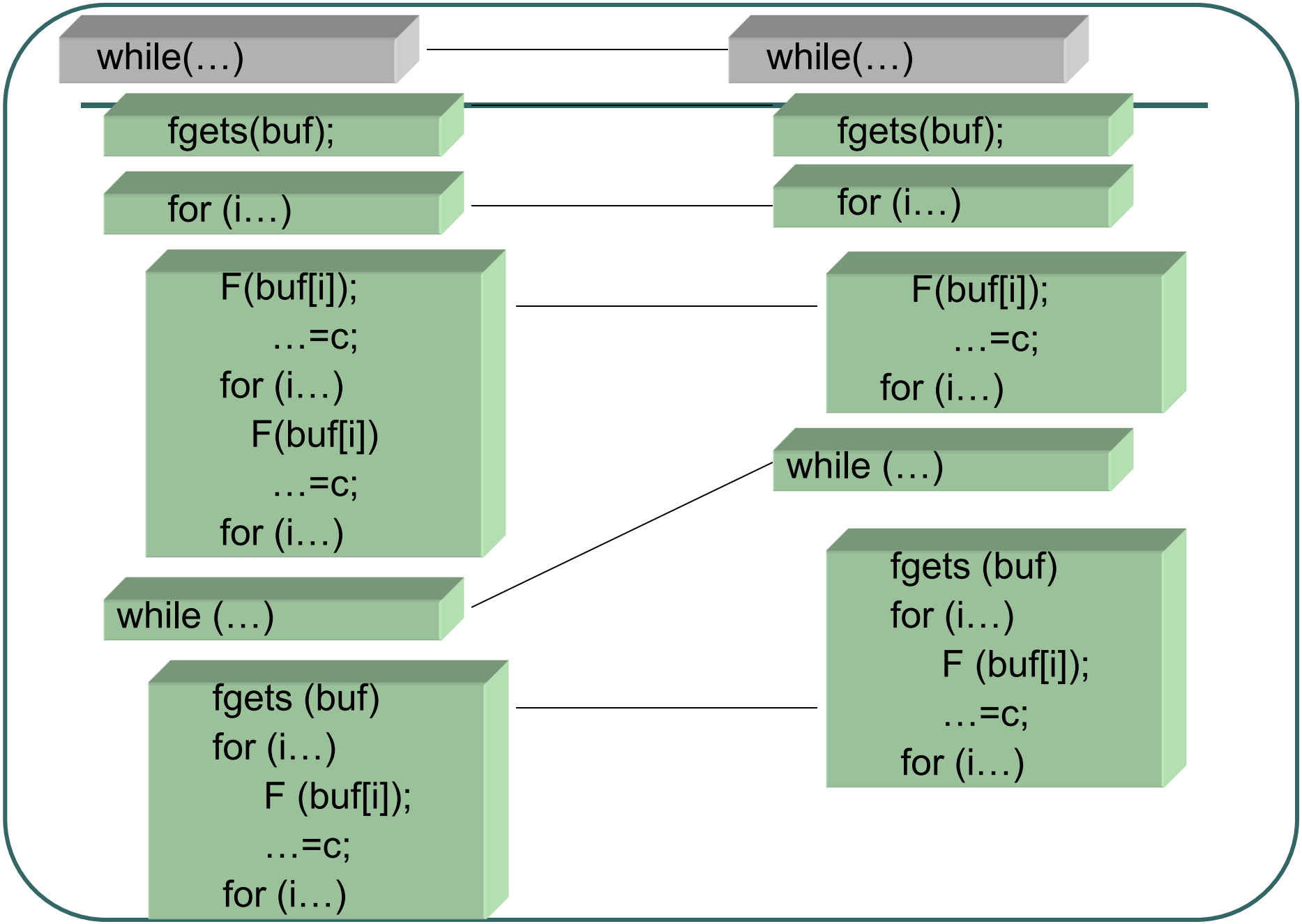
a\n

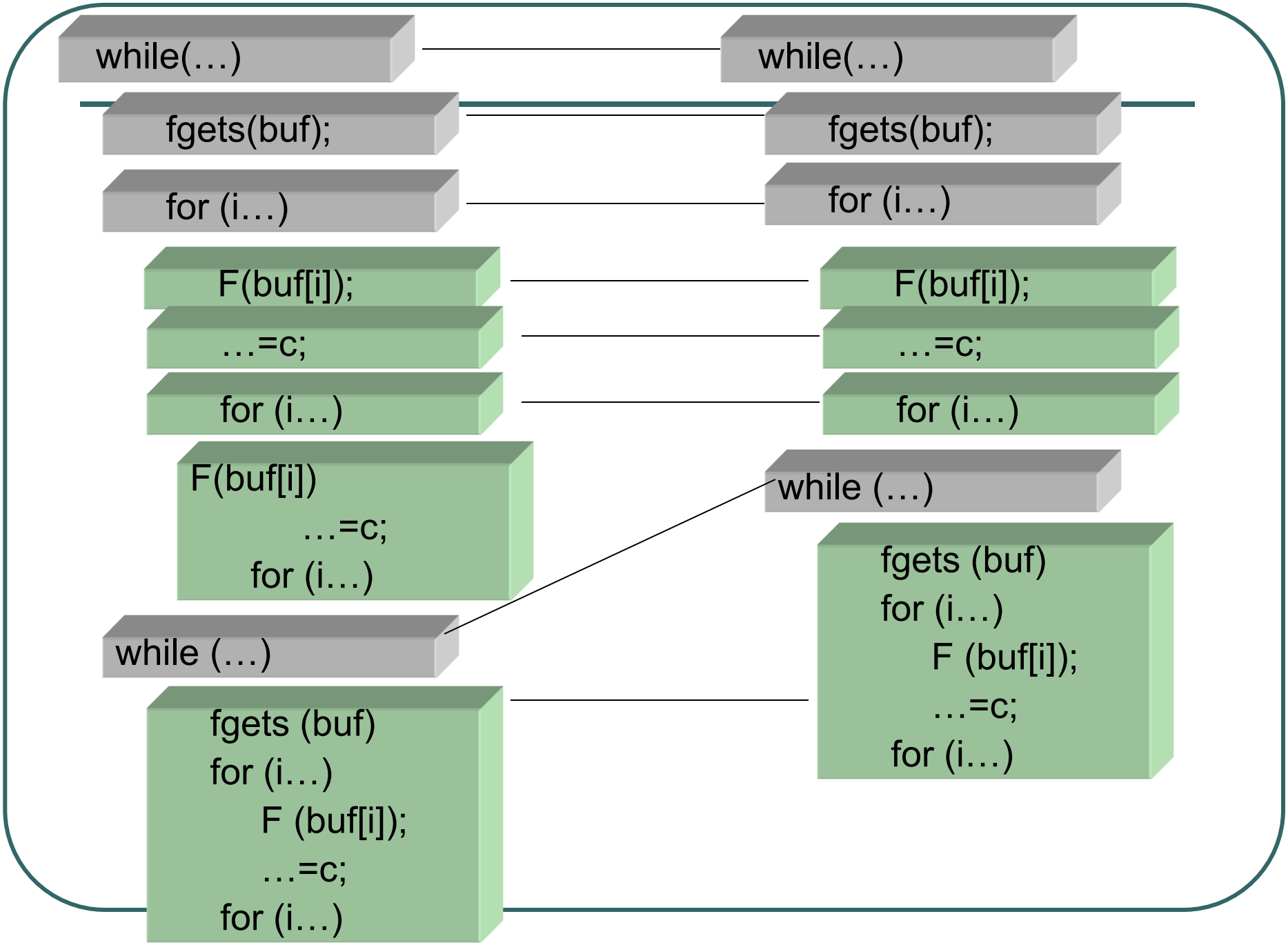
1

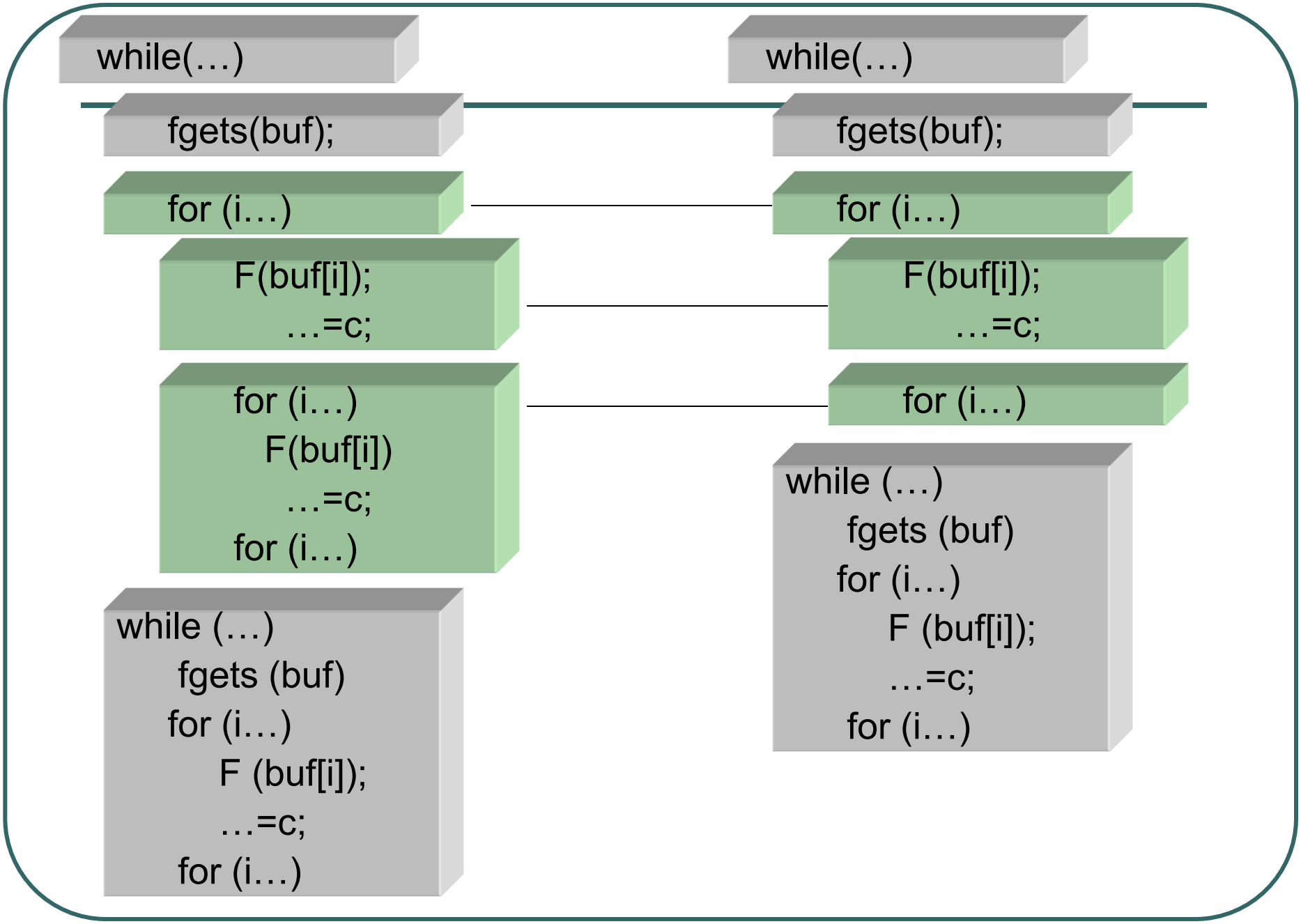
```
while(…)
    fgets(buf);
    for (i…)
        F(buf[i]);
            …=c;
    for (i…)
while (…)
    fgets (buf)
    for (i…)
        F (buf[i]);
            …=c;
    for (i…)
```

```
while(…)
    fgets(buf);
    for (i…)
       F(buf[i]);
          …=c;
    for (i…)
       F(buf[i])
          …=c;
    for (i…)
while (…)
    fgets (buf)
   for (i…)
        F (buf[i]);
        …=c;
    for (i…)
```

```
while(…)
    while(…)
       fgets(buf);
       for (i…)
          F(buf[i]);
             …=c;
       for (i…)
    while (…)
       fgets (buf)
      for (i…)
           F (buf[i]);
           …=c;
       for (i…)
```


while(…)
while(…)
fgets(buf);
fgets(buf);
for (i…)
for (i…)
F(buf[i]);
…=c;
for (i…)
F(buf[i])
…=c;
for (i…)
F(buf[i]);
…=c;
for (i…)
while (…)
while (…)
fgets (buf)
for (i…)
F (buf[i]);
…=c;
for (i…)
fgets (buf)
for (i…)
F (buf[i]);
…=c;
for (i…)


while(…)
while(…)
fgets(buf);
fgets(buf);
for (i…)
for (i…)
F(buf[i]);
F(buf[i]);
…=c;
…=c;
for (i…)
for (i…)
F(buf[i])
…=c;
for (i…)
while (…)
fgets (buf)
for (i…)
F (buf[i]);
…=c;
for (i…)
while (…)
fgets (buf)
for (i…)
F (buf[i]);
…=c;
for (i…)


while(…)
fgets(buf);
for (i…)
F(buf[i]);
…=c;
for (i…)
F(buf[i])
…=c;
for (i…)
while (…)
fgets (buf)
for (i…)
F (buf[i]);
…=c;
for (i…)
while(…)
fgets(buf);
for (i…)
F(buf[i]);
…=c;
for (i…)
while (…)
fgets (buf)
for (i…)
F (buf[i]);
…=c;
for (i…)

Formal Definition

- Execution description language (EDL).
  - Context free grammar;
  - Constructed automatically from the program;
  - Describes all possible executions;
    - An execution is a string accepted by the grammar.
  - Describes region nestings.

Execution Description Language

Program

```
1: while(..)
2:     s1;
3: s2;
4:     s2;
```

Executions

```
13
1 1213
1 121213
```

EDL

$L \rightarrow \underline{1} R_1 \underline{3}$

$R_1 \rightarrow \underline{21} R_1 \mid \varepsilon$

Grammar construction:

(1) A left hand side (non-terminal) symbol is generated for each predicate and function, representing a region;

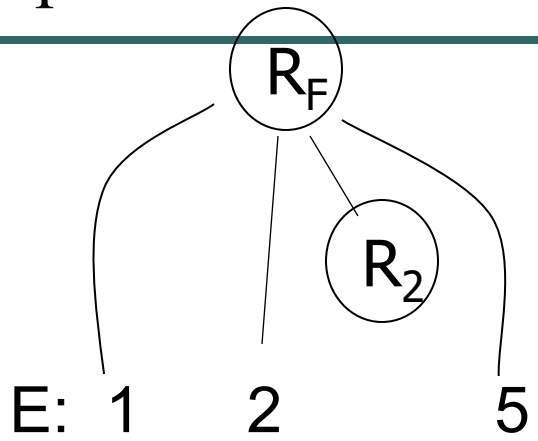
(2) The statements that are control dependent on the predicate (function entry) and their subregions constitute the right hand side.

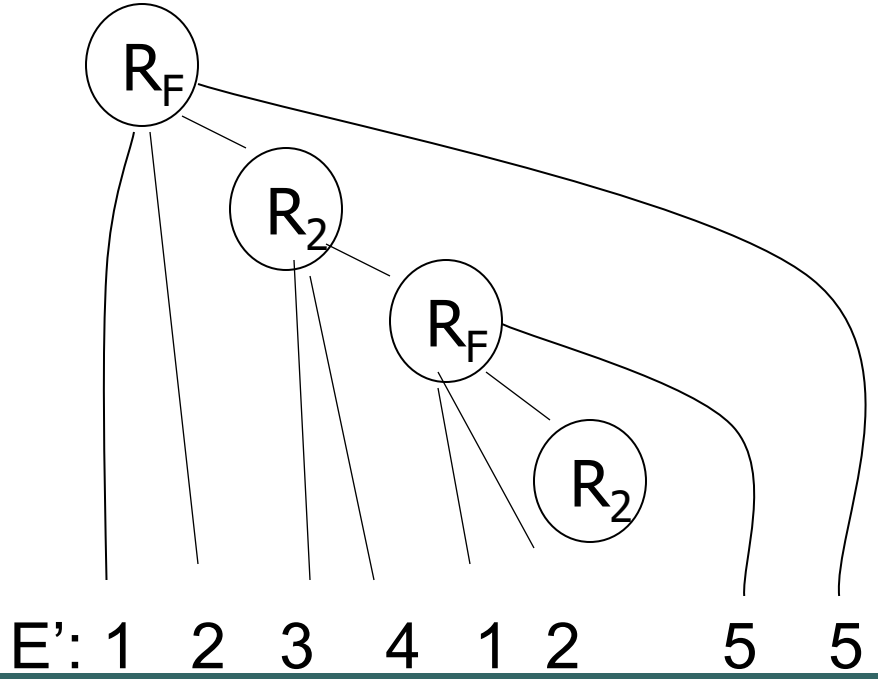
Example Revisit

```
F ( ) {
  1:  X = …
  2:  if (P) {
  3:      *p = *p …
  4:      F( )
      }
  5:  …= X
}
```

$R_F \rightarrow \underline{12} R_2 \underline{5}$

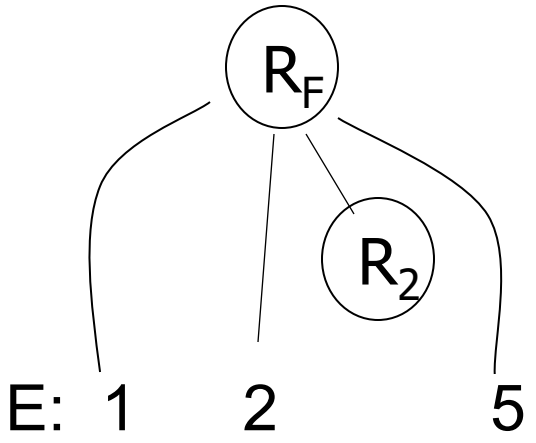
$R_2 \rightarrow \underline{34} R_F \mid \varepsilon$

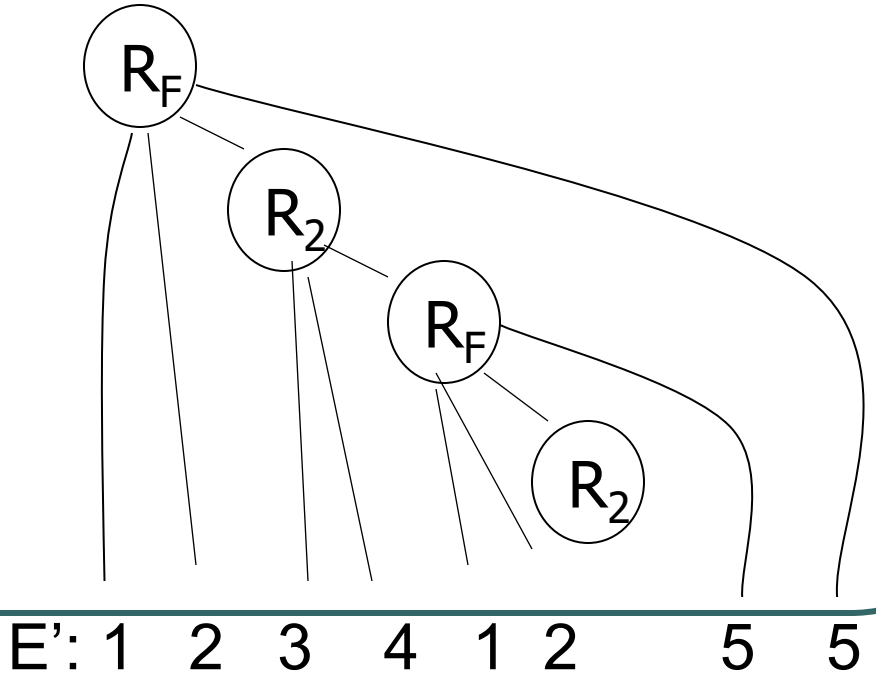




Execution Index

- Given an execution, the index of an execution point is the path in the EDL derivation tree that leads from the root to the execution point.
  - Reflect region nestings.
  - Two points in two respective executions align if they have the same index.





Basic Algorithm

- Compute and maintain indices online
  - The derivation tree is not explicitly built; only the current index, which is the current region nesting is maintained.
- Use a stack to maintain the index.
  - Essentially, it is very similar to control dependence stack (CDS)
    - Instrument at branches and their post-dominators, function calls and returns;
  - The entire stack stands for the index for the current execution point.

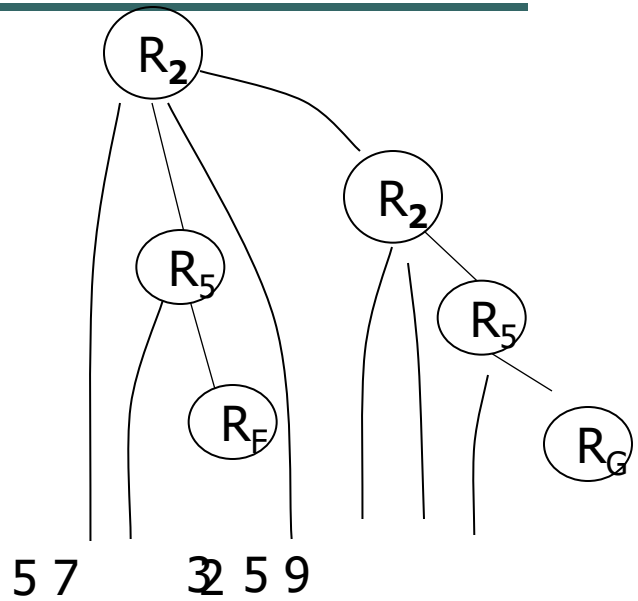
Semantic Augmentation

- Structural indexing only encodes control structures;
  - Not sufficient for some cases.
    - Event dispatch loop implementation;
    - Characterized by a switch inside a while.
- Provide a way to encode data in indices.
  - Based on programmer annotation.

Example

```
1: ...
2: while (..) {
3:     c=getc();
4:     …
5:     switch (c) {
6:     case 'a':
7:         F(c); break;
8:     case 'b':
9:         G(c); break;
10:    }
11: }
12:
```

Input: a b

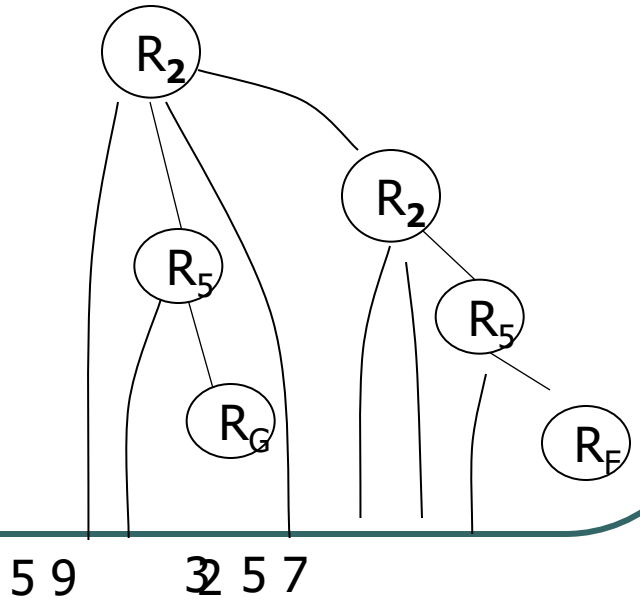


$R \rightarrow \underline{12} R_2$

$R_2 \rightarrow \underline{345}\ R_5\ \underline{2}\ R_2 \mid \varepsilon$

$R_5 \rightarrow \underline{7}\ R_F \mid \underline{9}\ R_G$

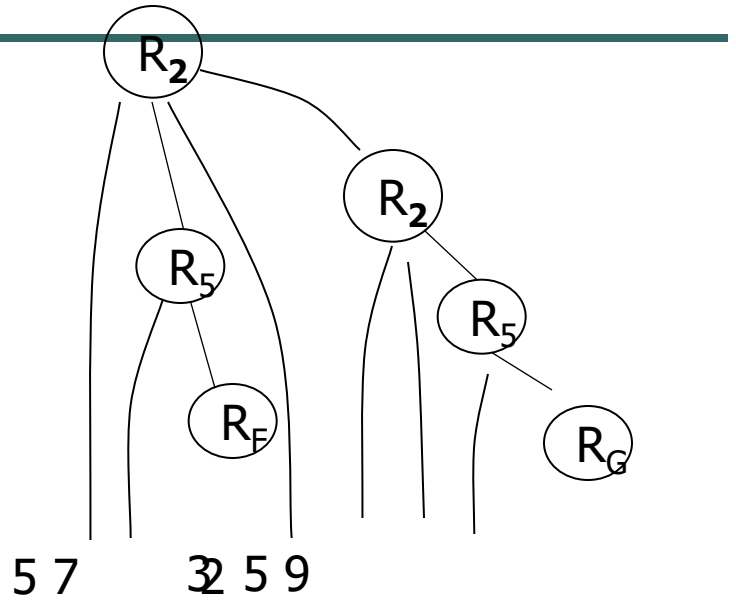
Input: b a



Example

```
1: ...
2: while (..) {
3:     c=getc();
4:     IDX_DATA(c);
5:     switch (c) {
6:     case 'a':
7:         F(c); break;
8:     case 'b':
9:         G(c); break;
10:    }
11: }
12:
```

Input: a b

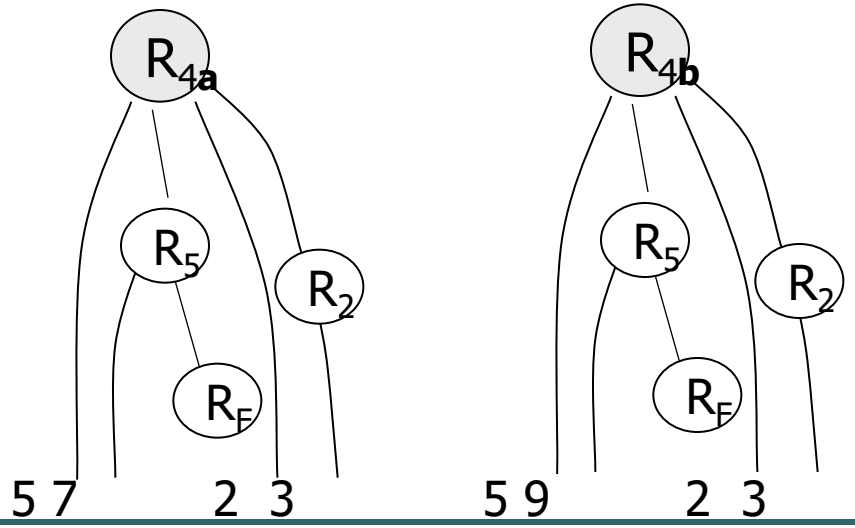


$R \rightarrow R_{4a} \mid R_{4b}$

$R_{4a} \rightarrow \underline{5}\ R_5\ \underline{2}\ R_2$

$R_2 \rightarrow \underline{3} \mid \varepsilon$

$R_5 \rightarrow \underline{7}\ R_F \mid \underline{8}\ R_G$



Applications

- Running the same program with the same input

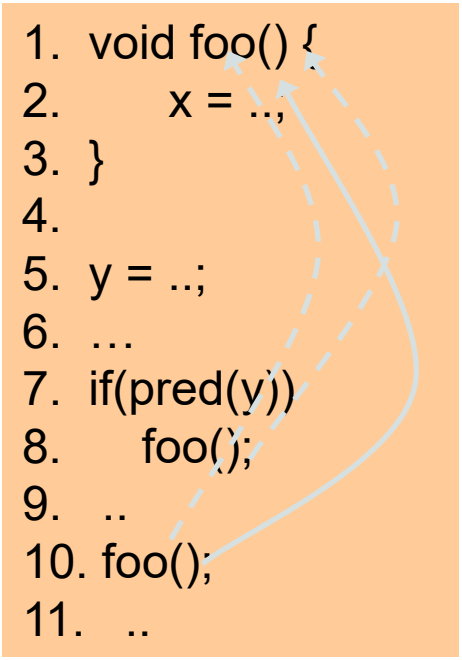
  -- Setting breakpoints in debugging

1. break at <R_{main}, R_{foo}, 1>

2. restart, “set y=<val>”

3. continue running.

```
1.  void foo() {
2.        x = ..;
3.  }
4.
5.  y = ..;
6.  …
7.  if(pred(y))
8.     foo();
9.   ..
10. foo();
11.   ..
```



Reading Assignment

- *Efficient Program Execution Indexing,* PLDI 2008.

Challenge Three (2 extra credits)

- *Heisenbugs* refer to those bugs that change or disappear once a debugger is used. They are notorious for being very difficult to debug. Classic solution is to use expensive tracing, which is prohibitively expensive for product runs. With the primitives of indexing and slicing, please sketch a plan to reproduce heisenbugs.
  - Assume heisenbugs are caused by thread concurrency.
  - You can assume indexing is supported in product runs so that when a failure occurs, the index of the failure can be reported.
  - The failure may not manifest itself in a simple re-execution. You may have to insert perturbations (e.g. synchronizations) to alter the timing of the original execution. The challenge is how and where to perturb.
  - A simple motivating example is expected
  - Limit to 2 pages.

Memory leaks

- What
  - Allocating memory without releasing later
- Why bad
  - Reduce performance
  - May cause crashes
- How to solve
  - Find out where exactly memory is leaked

Overview

- Kinds of memory leaks
- Existing tools
- Our solution
- False positives
- Evaluation
- Future Work

Physical Leaks

- What
  - Last pointer to a block of memory is destroyed
- Causes
  - Pointer overwrite
  - Free block containing pointer
  - Stack shrinks

Physical Leaks (2)

- Important information:
  - Allocation site of block
  - Location where last pointer is lost
  - Assignment location of last pointer

Logical Leaks

- What
  - Reachable pointer available, but block never freed
- Important information
  - Allocation site of block
  - Where are these pointers?

Existing Tools

- Most: where was leaked block allocated
  - Pro: simple, very fast
  - Con: may not be enough information
- Insure++: also where last pointer lost
  - Pro: more information for physical leaks
  - Con: not more info for logical leaks, source code instrumentation, no description of technique, expensive

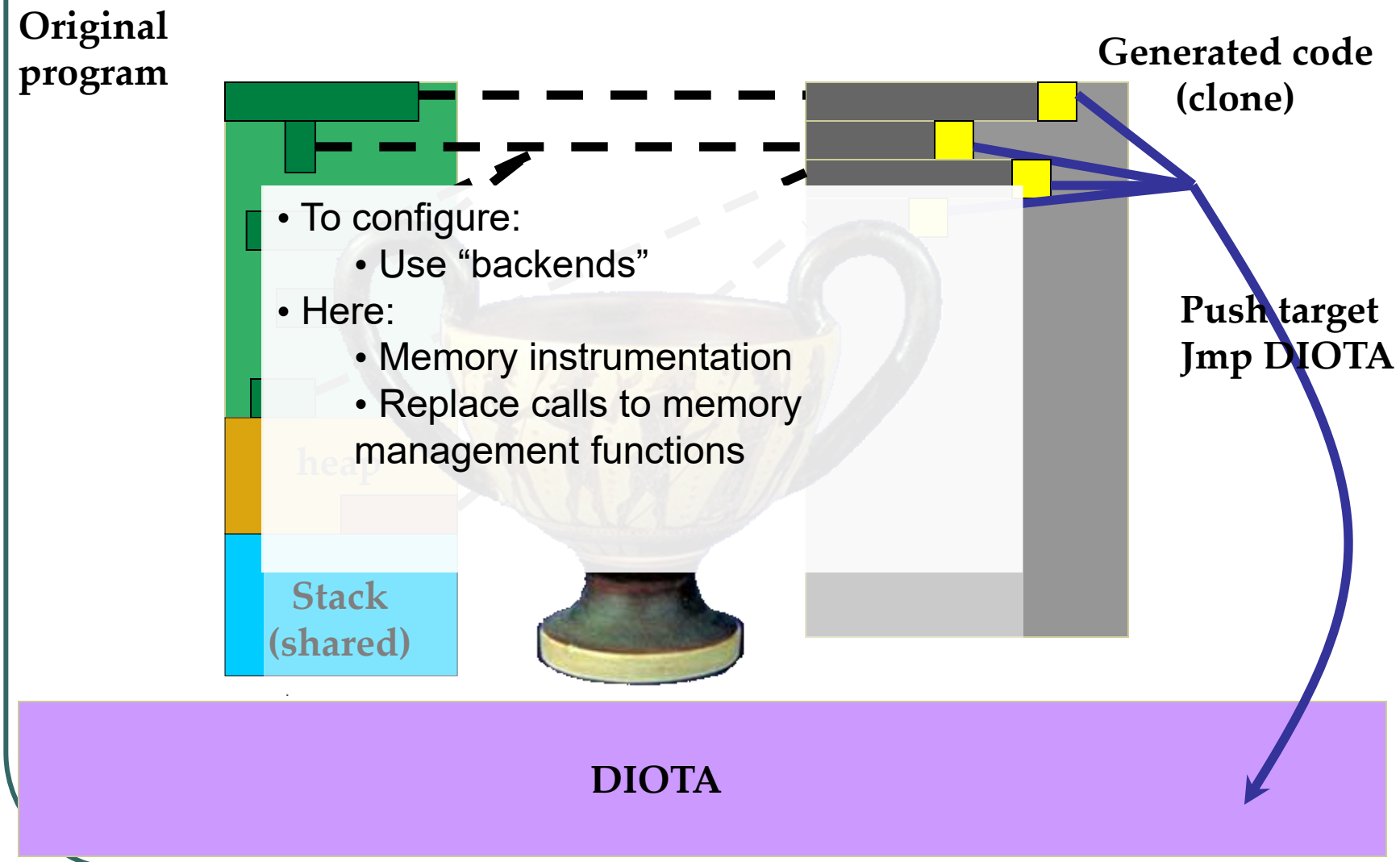
Our solution

- Keep track of allocated blocks
- Use dynamic instrumentation[1] to detect all references to memory blocks (DIOTA)

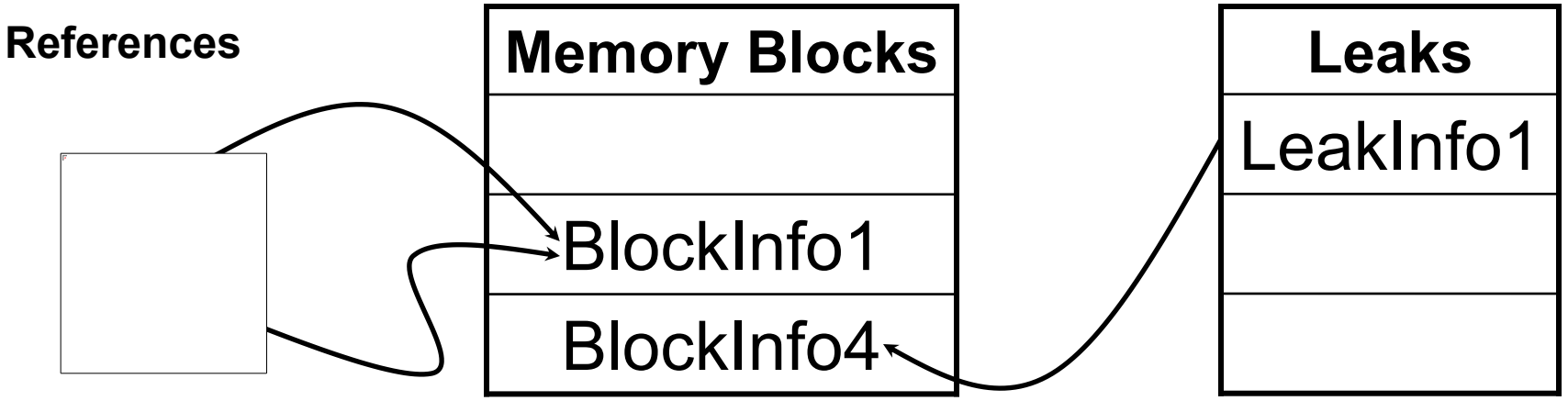
- Physical leak: reference count of block becomes zero
- Logical leak: unfreed blocks which are not physical leaks

[1] *Principle of "dynamic instrumentation" covered by EU software patent AT&T*

DIOTA principle



Overview Leak Detection



Tree:

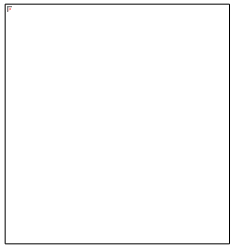
- Random and sequential access
- First element (stack)

Hashtable:

- Only random access necessary

Allocations

References



Memory Blocks
BlockInfo1

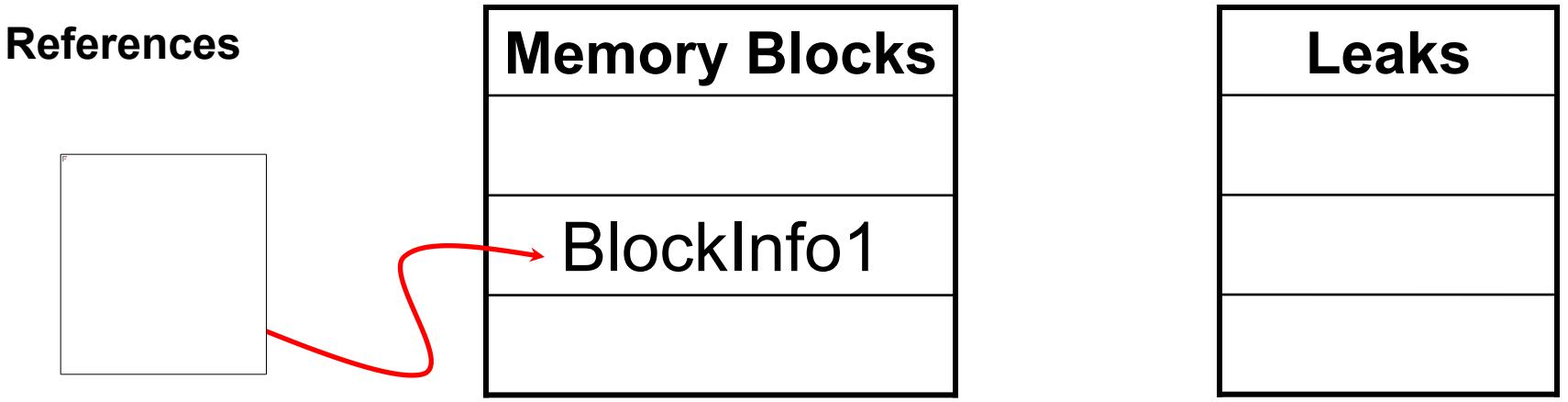
Leaks

```
void *a = malloc(32);
```

Reserved blocks:

- Intercept memory management functions
- Have reference count

References

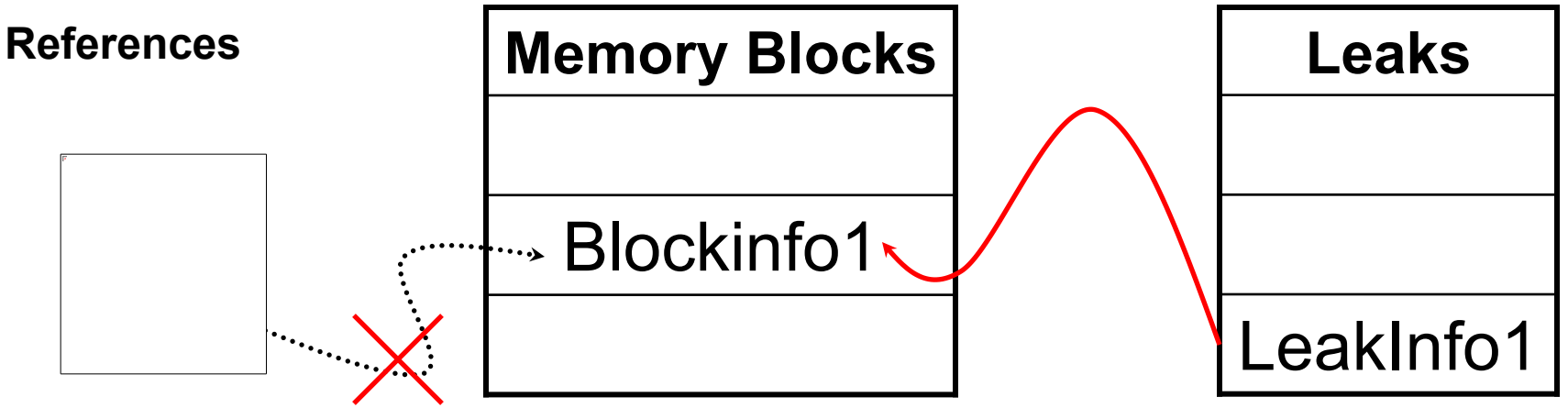



```
void *a = malloc(32);
```

Reference Detection:

- Results of stores
- Begin addresses of blocks

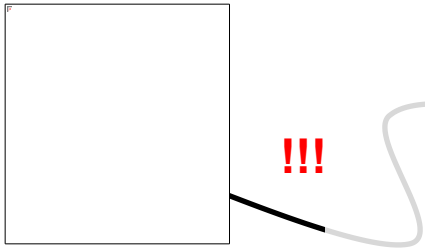
Leaks




```
void *a = malloc(32);
a = NULL;
```

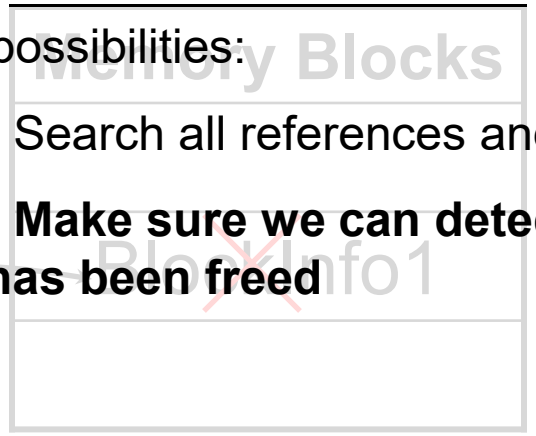
Stale References

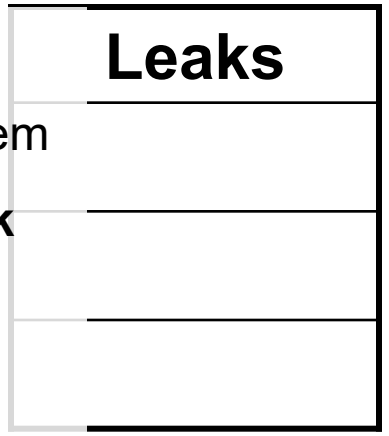
References



Two possibilities:

- Search all references and destroy them
- **Make sure we can detect that block has been freed**





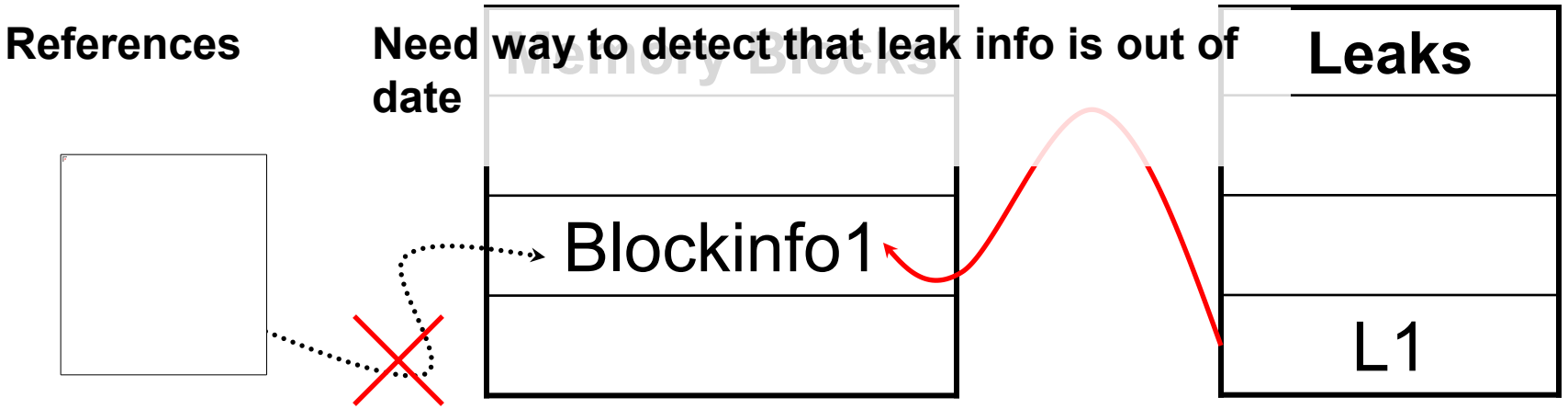

```
void *a = malloc(32);
free(a);
a = NULL;
```

Stale References (2)

- All BlockInfo's get a unique ID
- References get copy of ID
- When freeing a block, give it a new ID

- BlockInfo's come from dedicated pool
  - ➔ Can be reused as soon as block is freed
  - ➔ Direct pointer to BlockInfo from Reference

False Positives




```
void *a = malloc(32);
a += 2*sizeof(void*);
...
void *b = a-2*sizeof(void*);
...
```

False Positives (2)

- Every memory block gets a *usecount*
- *usecount* is increased at stores
- current *usecount* is copied to leakinfo
- Delay reporting of leaks:
  - BlockID's don't match: block freed
  - *usecount*s don't match: block still addressed later

Other issues

- C++: pointers to BlockStart+sizeof(void*)
- False negatives (random value == ptr)
- Circular structures

Evaluation

- Slowdown: 250 - 350 times in real world programs
- Example bug found using implementation:

```
1 for(int i=0;i<exp+1;i++) {
    set<map<..> > new_terms = find_terms(...);
    // fix memory leak found by DIOTA
    value_clear(result->arr[i].d);
5   value_clear(result->arr[i].x.n);
    result->arr[i] = translate_one_term(...);
  }
```

Software Fault Localization

- Develop a robust and reliable fault localization technique to identify faults from *dynamic behaviors* of programs
- Reduce the cost of program debugging by providing *a more accurate set of candidate fault positions*
- Provide software engineers with effective tool support

Perfect Bug Detection

- A bug in a statement will be detected by a programmer if the statement is examined
  - ~~A correct statement will not be mistakenly identified~~ as a faulty statement
  - If the assumption does not hold, a programmer may need to examine more code than necessary in order to find a faulty statement

140

Commonly Used Techniques

- Insert *print* statements
- Add *assertions* or set *breakpoints*
- Examine core dump or stack trace

Rely on programmers' intuition and domain expert knowledge

Software Fault Localization

- Execution Dice-based Fault Localization
- Suspiciousness Ranking-based Fault Localization
  - Program Spectra-based Fault Localization
  - Code Coverage-based Fault Localization
  - Statistical Analysis-based Fault Localization
  - Neural Network-based Fault Localization
  - Similarity Coefficient-based Fault Localization
- Mutation-based Automatic Bug Fixing

142

Homework 3

Lựa chọn 1 công cụ/ngôn ngữ lập trình mà nhóm thống nhất là nắm vững nhất.

1) Minh họa ý tưởng phân tích chương trình sử dụng Tracing

2) Minh họa ý tương phân tích chương trình sử dụng kỹ thuật Dynamic Slicing

3) Minh họa các kỹ thuật liên quan đến execution indexing

4) Minh họa các kỹ thuật liên quan đến fault localization