DELTAPATH: PRECISE AND SCALABLE CALLING CONTEXT ENCODING

Qiang Zeng*, **Junghwan Rhee**, Hui Zhang, Nipun Arora,
Guofei Jiang, Peng Liu*

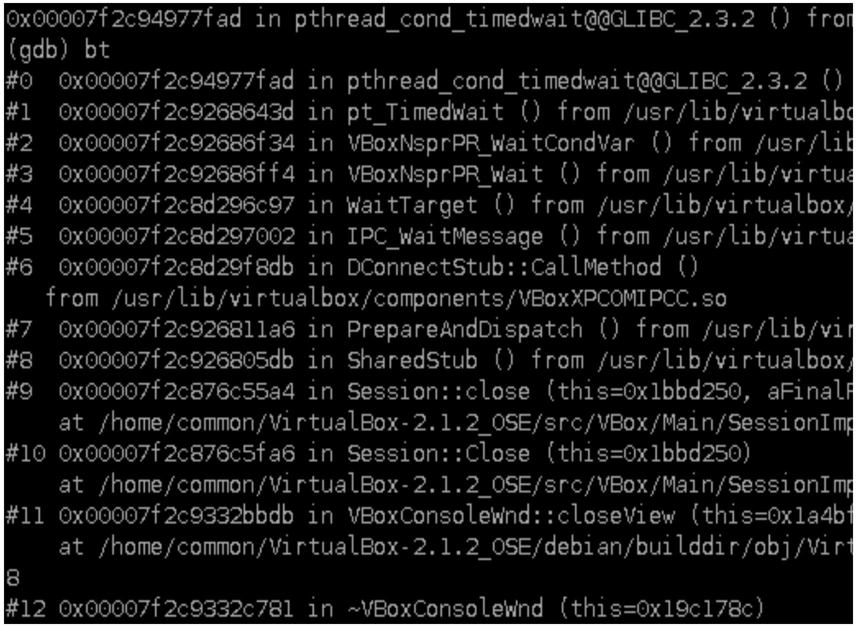
NEC Laboratories America
*Penn State University

NEC Laboratories
America
Relentless passion for innovation
www.nec-labs.com

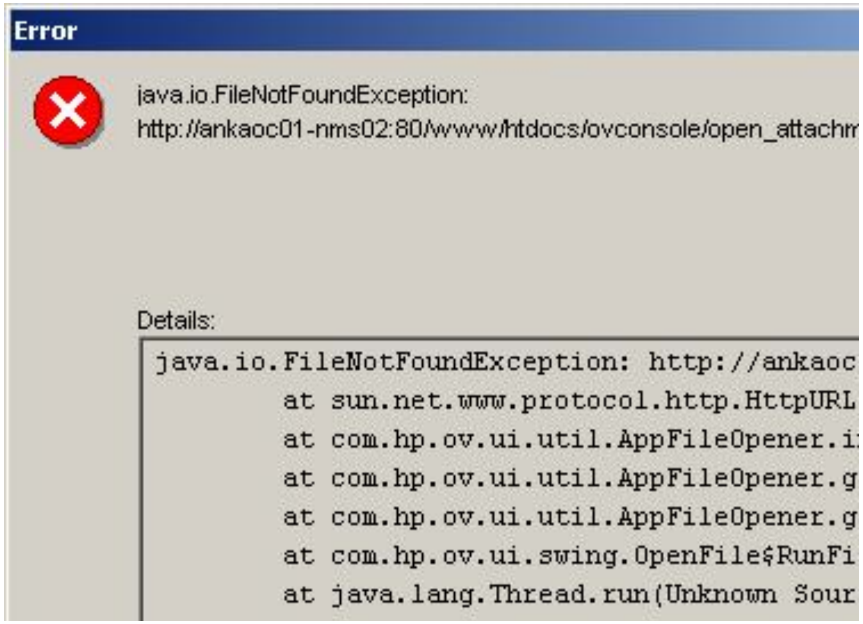
PENNSTATE
1 8 5 5

Calling Context

- Calling Context is a sequence of active function/method invocations that lead to a program location (i.e., call stack status).



```
0x00007f2c94977fad in pthread_cond_timedwait@@GLIBC_2.3.2 () fron
(gdb) bt
#0  0x00007f2c94977fad in pthread_cond_timedwait@@GLIBC_2.3.2 ()
#1  0x00007f2c9268643d in pt_TimedWait () from /usr/lib/virtualbo
#2  0x00007f2c92686f34 in VBoxNsprPR_WaitCondVar () from /usr/lib
#3  0x00007f2c92686ff4 in VBoxNsprPR_Wait () from /usr/lib/virtua
#4  0x00007f2c8d296c97 in WaitTarget () from /usr/lib/virtualbox/
#5  0x00007f2c8d297002 in IPC_WaitMessage () from /usr/lib/virtua
#6  0x00007f2c8d29f8db in DConnectStub::CallMethod ()
   from /usr/lib/virtualbox/components/VBoxXPCOMIPCC.so
#7  0x00007f2c926811a6 in PrepareAndDispatch () from /usr/lib/vir
#8  0x00007f2c926805db in SharedStub () from /usr/lib/virtualbox/
#9  0x00007f2c876c55a4 in Session::close (this=0x1bbd250, aFinalR
    at /home/common/VirtualBox-2.1.2_OSE/src/VBox/Main/SessionImp
#10 0x00007f2c876c5fa6 in Session::Close (this=0x1bbd250)
    at /home/common/VirtualBox-2.1.2_OSE/src/VBox/Main/SessionImp
#11 0x00007f2c9332bbdb in VBoxConsoleWnd::closeView (this=0x1a4bf
    at /home/common/VirtualBox-2.1.2_OSE/debian/builddir/obj/Virt
8
#12 0x00007f2c9332c781 in ~VBoxConsoleWnd (this=0x19c178c)
```



Error

java.io.FileNotFoundException:
http://ankaoc01-nms02:80/www/htdocs/ovconsole/open_attachm

Details:

```
java.io.FileNotFoundException: http://ankaoc
        at sun.net.www.protocol.http.HttpURL
        at com.hp.ov.ui.util.AppFileOpener.i
        at com.hp.ov.ui.util.AppFileOpener.g
        at com.hp.ov.ui.util.AppFileOpener.g
        at com.hp.ov.ui.swing.OpenFile$RunFi
        at java.lang.Thread.run(Unknown Sour
```

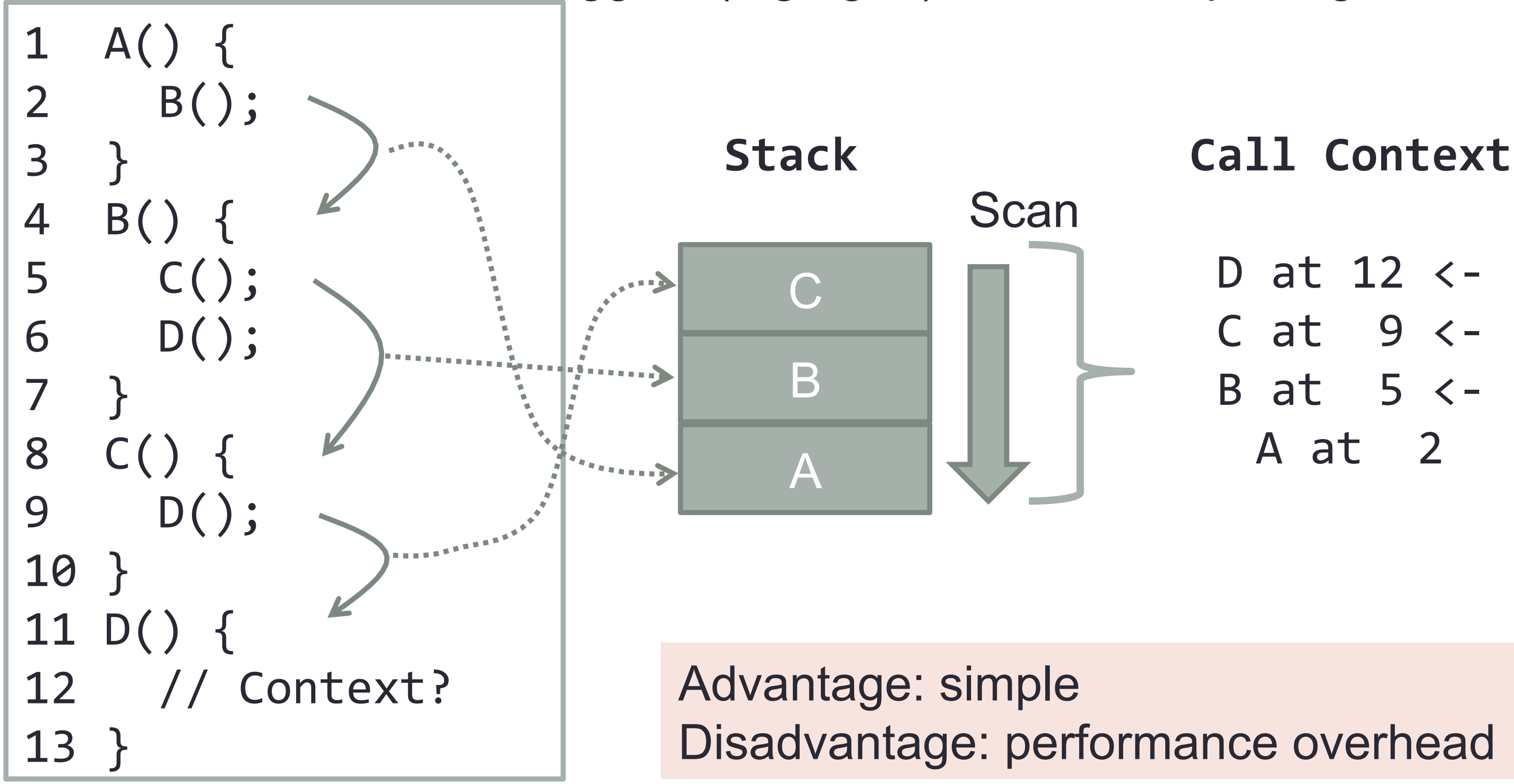
- Wide range of applications
  - Debugging, event logging, error reporting, testing, anomaly detection, performance optimization, profiling, security.

How to Collect Calling Contexts?

- Stack Walking
- Probabilistic Calling Context [OOPSLA'07]
- Precise Calling Context Encoding [ICSE'10]

Stack Walking

- Walk stack and collect context
  - Stack walking collects a set of return addresses from the stack.
  - Commonly used in debuggers (e.g., gdb) and error reporting



```
1  A() {
2    B();
3  }
4  B() {
5    C();
6    D();
7  }
8  C() {
9    D();
10 }
11 D() {
12   // Context?
13 }
```

Call Context

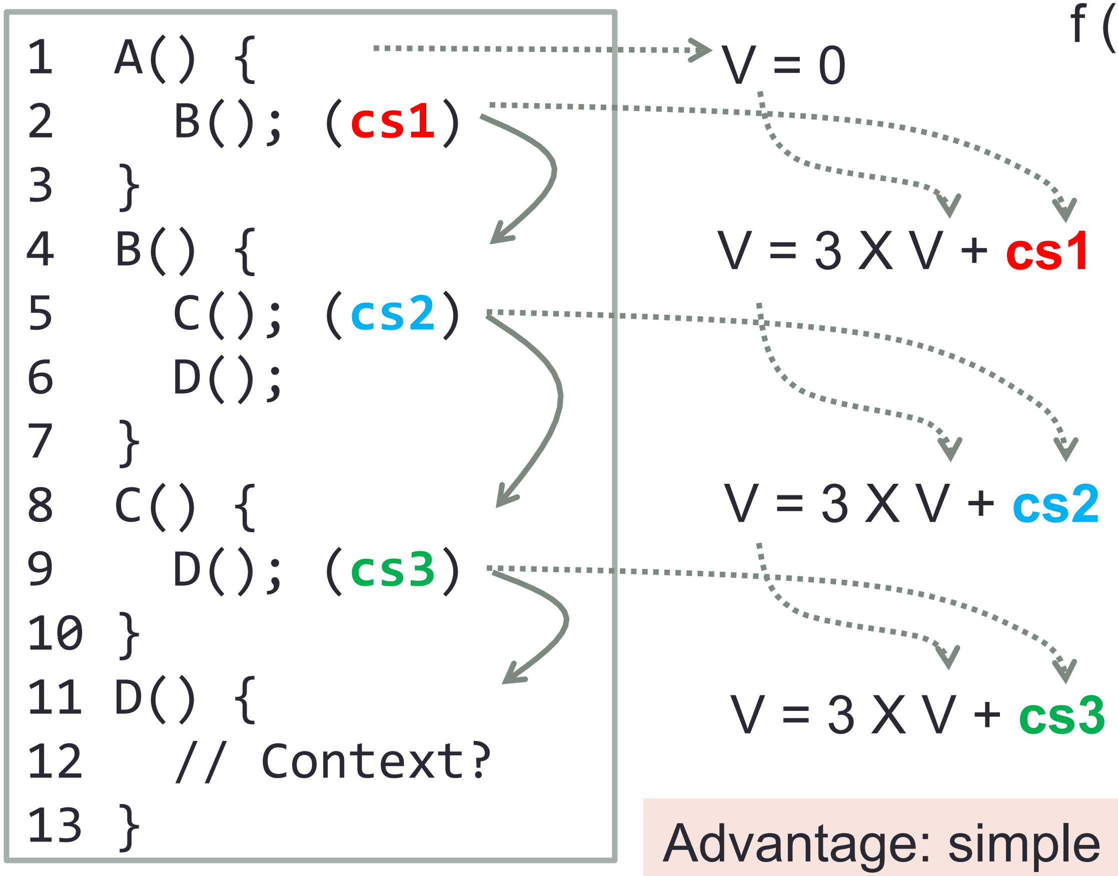
```
 D at 12 <-
 C at  9 <-
 B at  5 <-
   A at  2
```

Advantage: simple
Disadvantage: performance overhead

Probabilistic Calling Context [OOPSLA '07]

- Compute probabilistic calling context at runtime

$f(V, cs) := 3 \times V + cs$



```
A() {
   B(); (cs1)
}
B() {
   C(); (cs2)
   D();
}
C() {
   D(); (cs3)
}
D() {
   // Context?
}
```

$V = 0$

$V = 3 \times V + cs1$

$V = 3 \times V + cs2$

$V = 3 \times V + cs3$

Advantage: simple & fast encoding scheme
Disadvantage: decoding is not guaranteed.

Precise Calling Context Encoding [ICSE'10]

- Use unique numbering to represent a path in a CFG

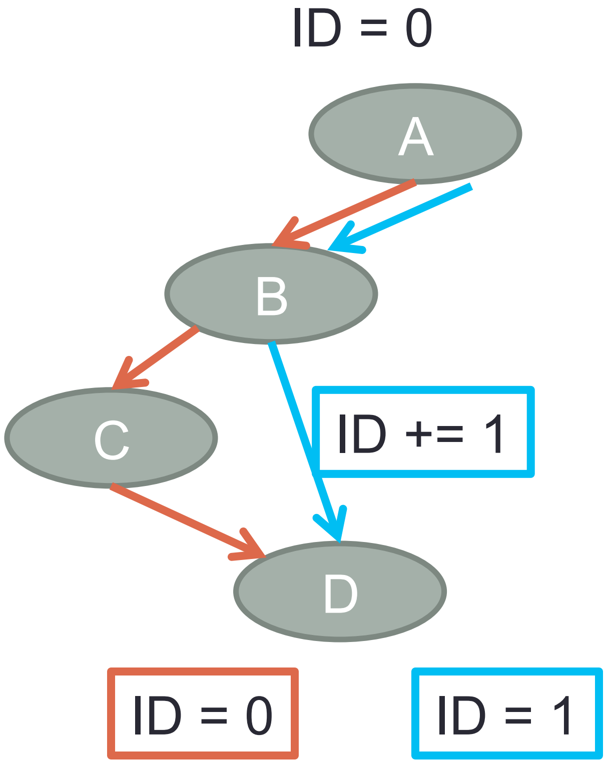
```
A() {
   B(); (cs1)
}
B() {
   C(); (cs2)
   D();
}
C() {
   D(); (cs3)
}
D() {
   // Context?
}
```

ID = 0

ID += 1

ID -= 1

ID ?



Advantage: Precise call context encoding and decoding

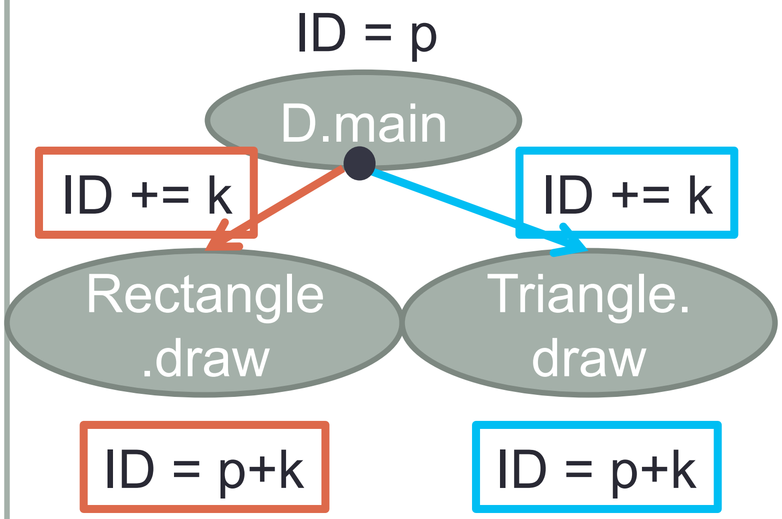
Precise Calling Context Encoding

```
class Shape { void draw() {}; }
class Rectangle extends Shape {
  void draw() {}
}
class Triangle extends Shape {
  void draw() {}
}
class D {
 static void main() {
 Shape a;
 if (input) a = new Rectangle()
 else a = new Triangle();
 a.draw();
   }
}
```

ID+=k

ID-=k

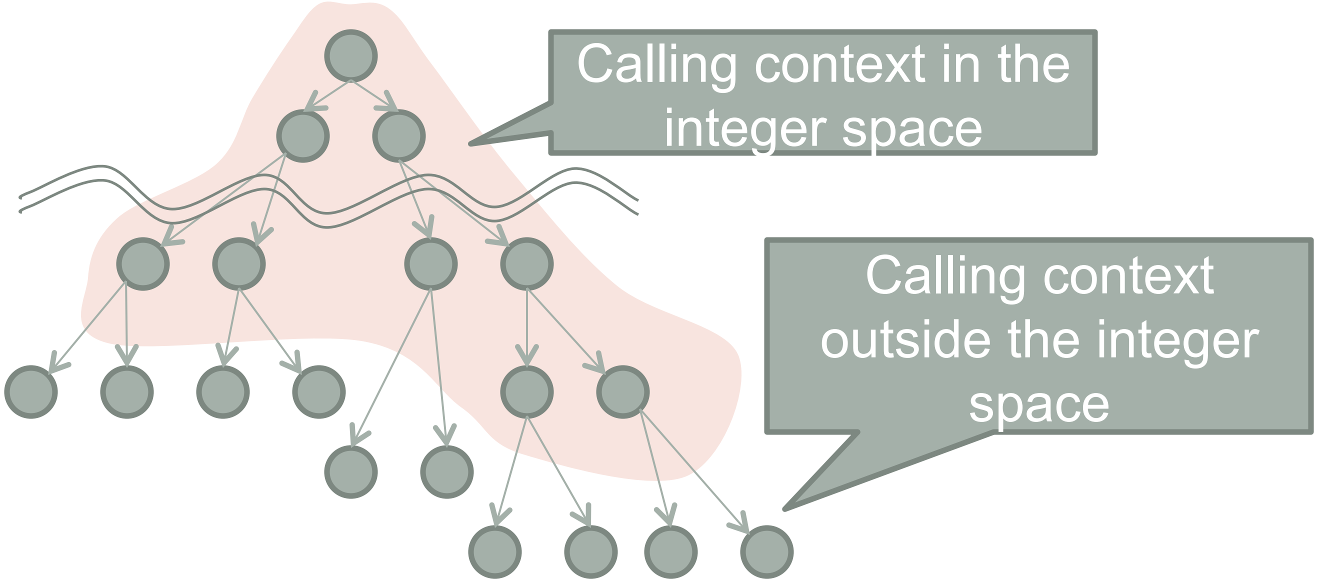
Dynamic dispatch
a call site can call either Rectangle.draw() or Triangle.draw()



Disadvantage 1: dynamic dispatch in object-oriented programs

Precise Calling Context Encoding

- PCCE maps each unique context into an integer.
- The integer space is insufficient for large programs.
  - Object oriented programs tend to have many small functions leading to a large context space.



Calling context in the integer space

Calling context outside the integer space

Disadvantage 2: PCCE addresses this problem using profiling and identifying hot and cold edges.

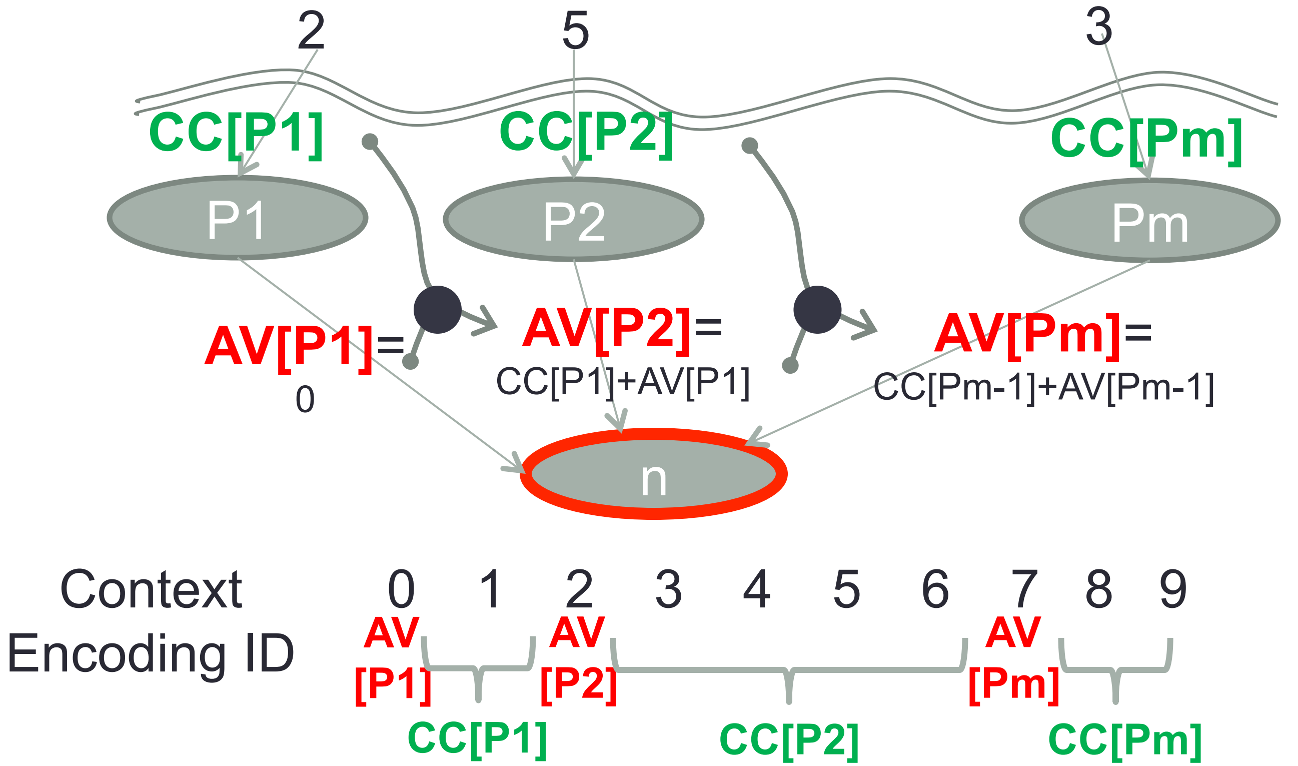
DeltaPath Features

- New precise and scalable calling context encoding algorithm for both procedural and object oriented programs
  - Overcome dynamic dispatch
  - Address encoding space pressure systematically

- Practical Issues
  - Dynamic class loading is handled.
  - Flexible encoding scope

Technique – Inflated Calling Context

- Basic properties of Precise Calling Context Encoding
  - Ensure the invariant that for a given node, its encoding space is divided into disjoint sub-ranges for unique numbering.
  - AV : addition value, CC : calling context count

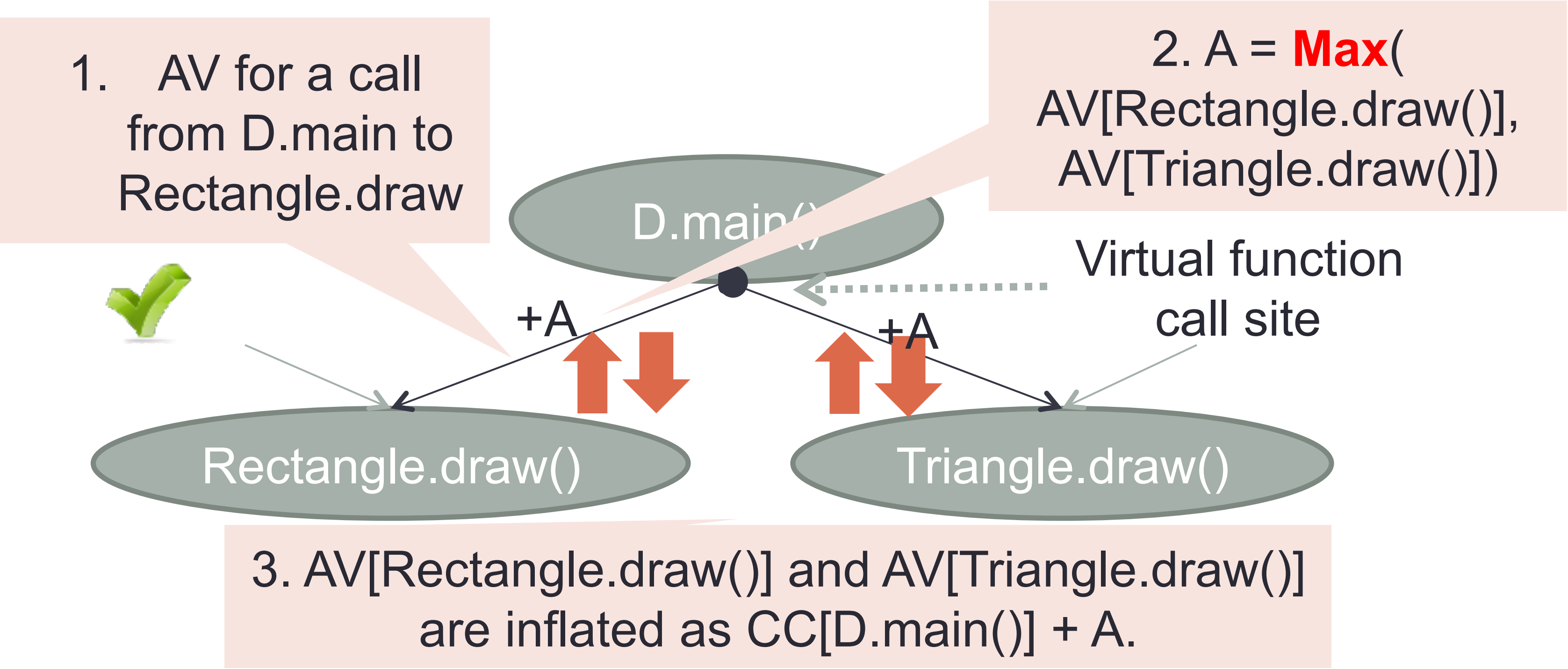


Invariants:
AV[Pi] = CC[Pi-1]+AV[Pi-1]
for i = 2, …, m
CC[n] >= CC[Pm] + AV[Pm]

Encoding ID space is partitioned using AV and CC

Technique – Inflated Calling Context

- Idea: **Inflated Calling Context**
  - While PCCE processes the nodes one by one, DeltaPath needs to take into account the current addition value for another node so that all nodes involved in dynamic dispatch can agree on the common addition value. This is achieved by the **inflation of calling context**.



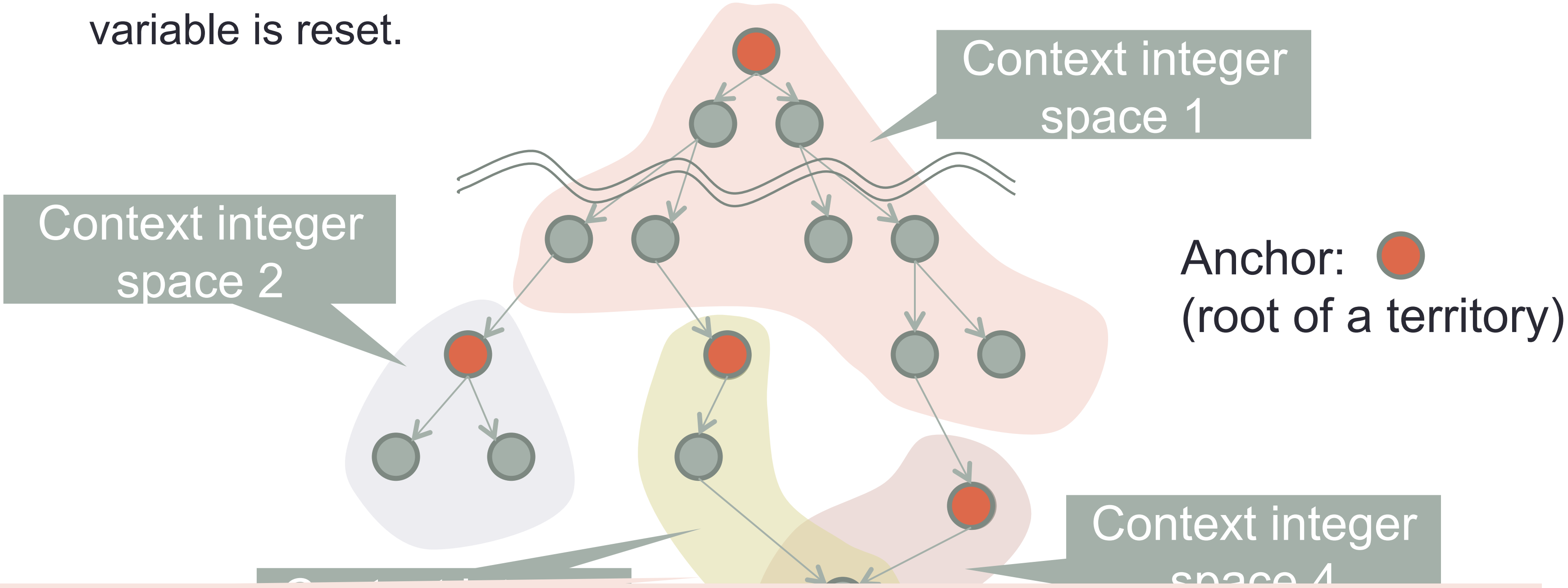
1. AV for a call from D.main to Rectangle.draw

2. A = **Max**(AV[Rectangle.draw()], AV[Triangle.draw()])

3. AV[Rectangle.draw()] and AV[Triangle.draw()] are inflated as CC[D.main()] + A.

Technique – Resolving Context Explosion

- Encoding for large-scale object-oriented programs
  - Systematically divides the CFG into **territories** whose contexts fit the limit of integer space.
  - On the detection of overflow, the node is added into the set of **anchor** nodes and static analysis is restarted (iterative approach).
  - At runtime an anchor flushes current context onto stack and the context variable is reset.



Context integer space 1

Context integer space 2

Context integer space 4

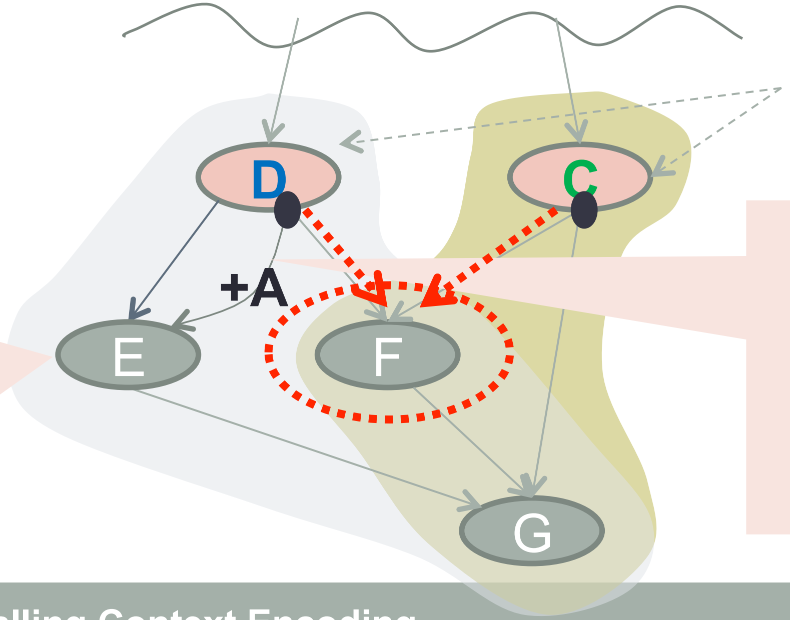
Anchor: ●
(root of a territory)

Challenge: The problem would be simple if each territory has only one entry point but in fact there are ***cross-territory calls***.

Technique – Resolving Context Explosion

- Multiplexing the contexts of multiple territories
  - The common addition value is used for all multiplexed territories. Thus the context variable should afford the context of all multiplexed territories.
  - Use **two dimensional states** in the algorithm to track contexts from multiple overlapped territories.
  - Use **inflation** to meet the invariants for multiple territories simultaneously.

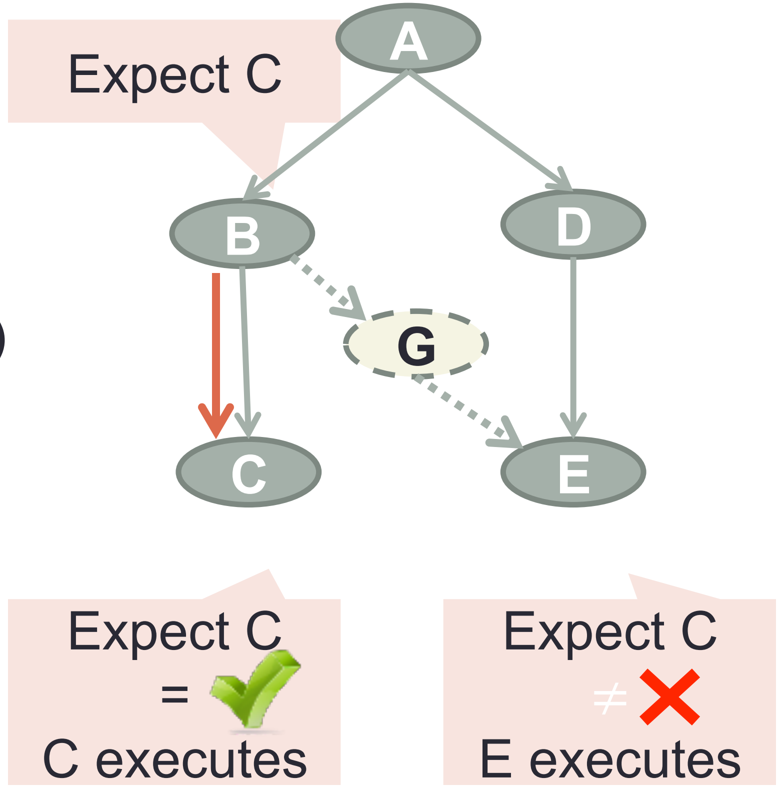
ICC[**node**][**anchor**]
CAV[**node**][**anchor**]
= inflated calling context count and addition value
at the **node** relative to the **anchor**



A = **Max**(
CAV[E][**D**],
CAV[F][**D**],
CAV[F][**C**])

Practical Issues

- Dynamic Class Loading
  - Java loads and combines code at runtime. Such code cannot be pre-analyzed causing **unexpected call paths (UCPs)**.
- Solution: Calling Context Tracking
  - We adopted control flow integrity (CFI) technique to detect UCPs.
  - For each call site, finds out the dispatch target nodes. Merge the sets that contain any overlap and assign unique set identifiers (SID).
  - Expected SIDs are stored at callers and checked at callees.



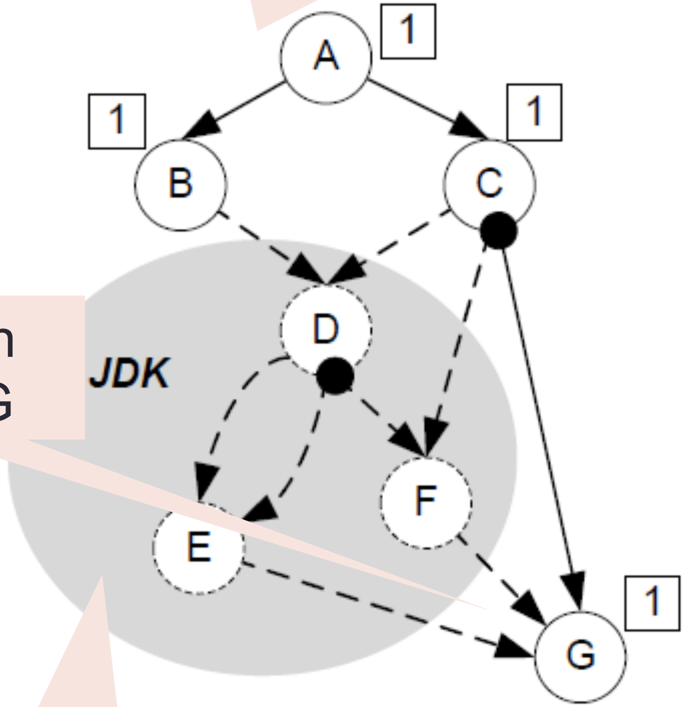
Expect C

Expect C = ✓ C executes

Expect C ≠ ✗ E executes

Practical Issues

- Do we need to track all code?
  - Java has large library code base which may be little of interest for debugging etc.
  - PCC redefines application only calling context and encodes it.
  - Also including all code inevitably will slow down execution.
- Solution: Flexible Encoding
  - Leveraging call path tracking we can skip encoding components of little interest the same way we handle dynamically loaded classes.
  - Call paths through skipped nodes are detected as UCPs.



Application code fully covered

UCPs on B/C -> G

No overhead in numerous libraries

Implementation and Evaluation

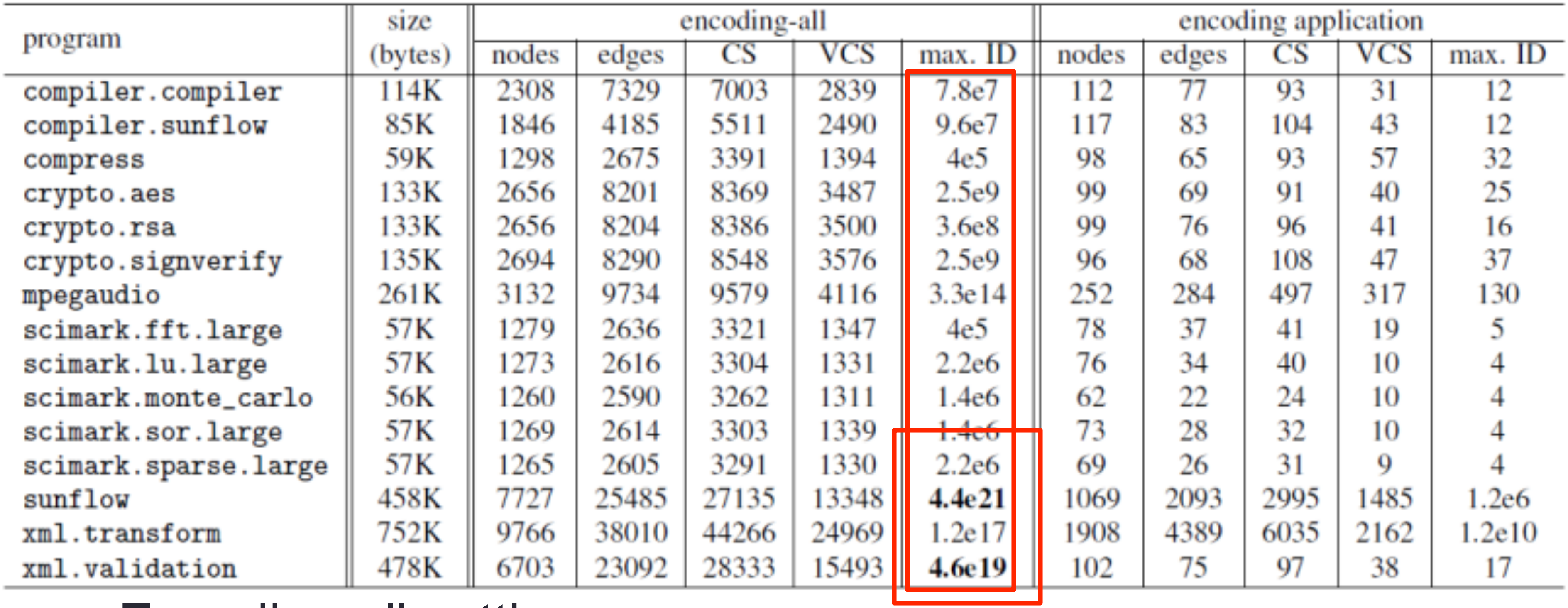
- Static Analysis
  - WALA (T.J. Watson Libraries for Analysis)
  - Analysis: Context Insensitive Control Flow Analysis (0-CFA)
  - Input: Binary only, No source code

- Runtime Module and Dynamic Instrumentation
  - A Java agent based on Javassist
  - Support Sun JVM (Version >= JDK 5.0)

- Evaluation
  - SPECjvm2008 Benchmark Suite
  - Intel Core i7 CPU, 8GB RAM
  - Ubuntu Linux 10.04
  - Sun JDK 1.6.0.24

Evaluation

- Static Program Characteristics

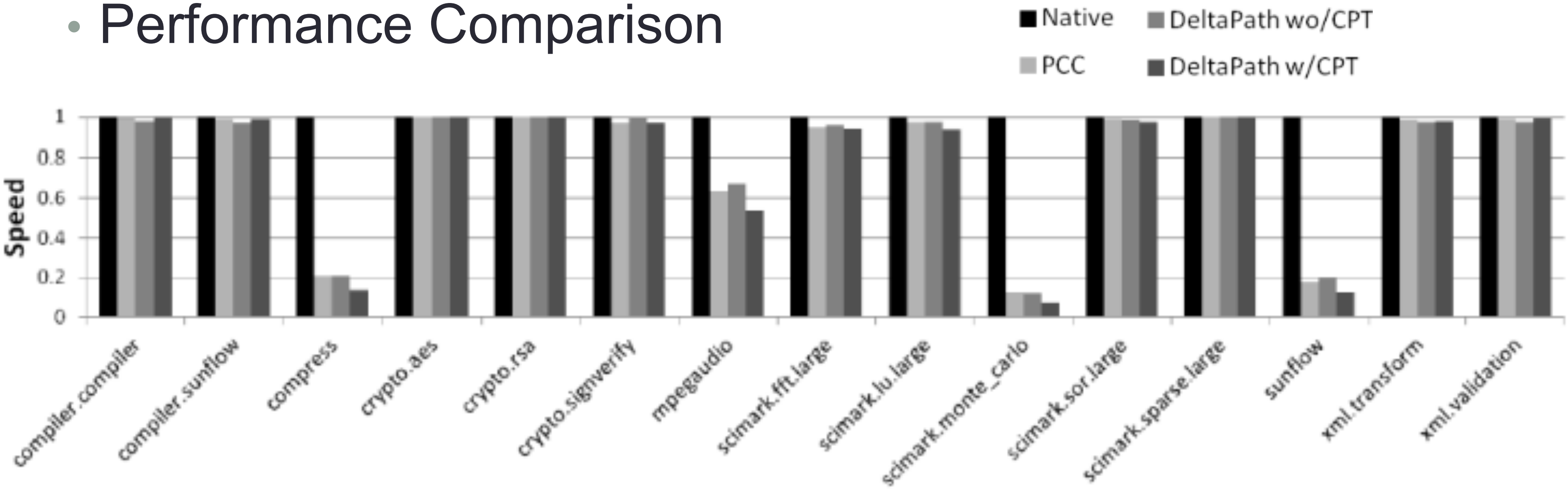


program	size (bytes)	encoding-all					encoding application				
		nodes	edges	CS	VCS	max. ID	nodes	edges	CS	VCS	max. ID
compiler.compiler	114K	2308	7329	7003	2839	7.8e7	112	77	93	31	12
compiler.sunflow	85K	1846	4185	5511	2490	9.6e7	117	83	104	43	12
compress	59K	1298	2675	3391	1394	4e5	98	65	93	57	32
crypto.aes	133K	2656	8201	8369	3487	2.5e9	99	69	91	40	25
crypto.rsa	133K	2656	8204	8386	3500	3.6e8	99	76	96	41	16
crypto.signverify	135K	2694	8290	8548	3576	2.5e9	96	68	108	47	37
mpegaudio	261K	3132	9734	9579	4116	3.3e14	252	284	497	317	130
scimark.fft.large	57K	1279	2636	3321	1347	4e5	78	37	41	19	5
scimark.lu.large	57K	1273	2616	3304	1331	2.2e6	76	34	40	10	4
scimark.monte_carlo	56K	1260	2590	3262	1311	1.4e6	62	22	24	10	4
scimark.sor.large	57K	1269	2614	3303	1339	1.4e6	73	28	32	10	4
scimark.sparse.large	57K	1265	2605	3291	1330	2.2e6	69	26	31	9	4
sunflow	458K	7727	25485	27135	13348	**4.4e21**	1069	2093	2995	1485	1.2e6
xml.transform	752K	9766	38010	44266	24969	1.2e17	1908	4389	6035	2162	1.2e10
xml.validation	478K	6703	23092	28333	15493	**4.6e19**	102	75	97	38	17

- Encoding all setting
  - 13 out of 15 need encoding space larger than a million
  - Two benchmarks have overflow of the 64bit integer (1.8 X 10^19).
  - Overflow is resolved by introducing 6~7 anchor nodes.

Evaluation

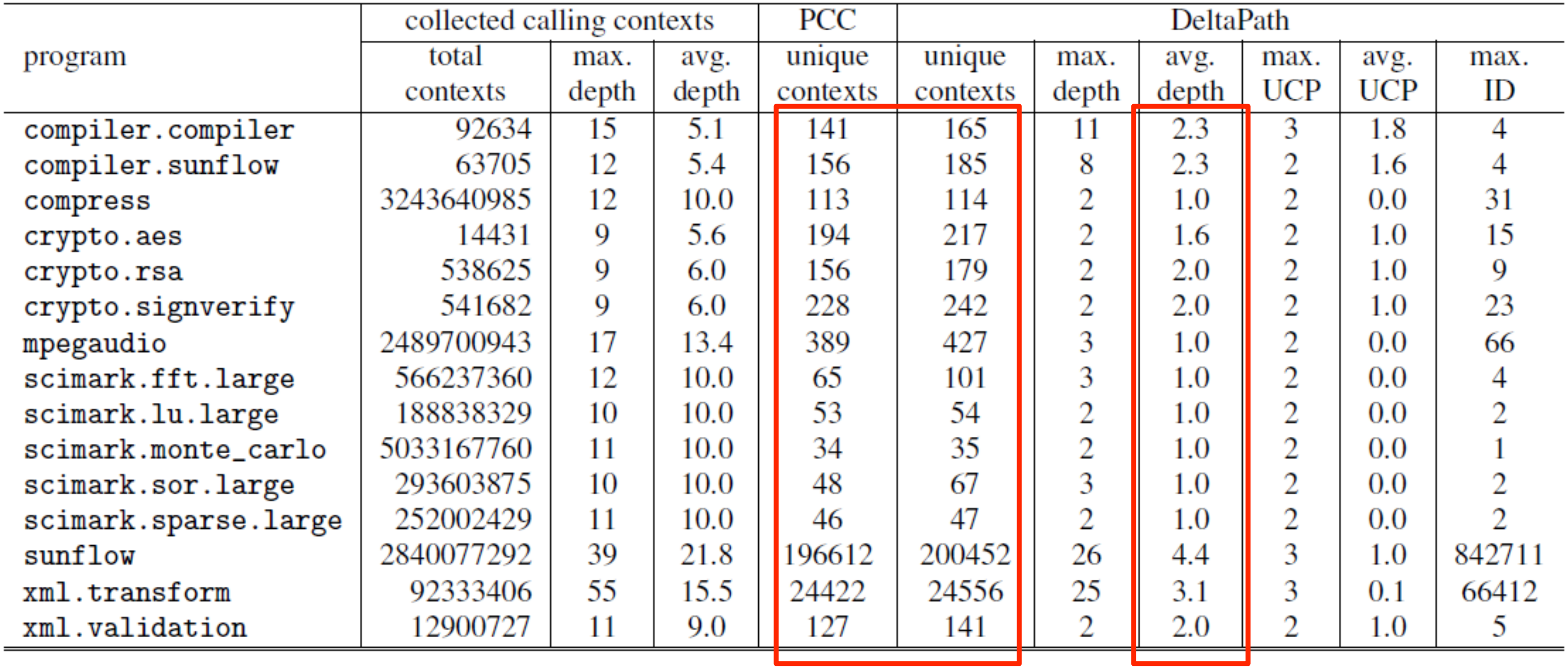
- Performance Comparison



- DeltaPath without Call Path Tracking : 32.51% (geometric mean)
- Call Path Tracking adds extra 6.79% slow down.
- Comparable with PCC (0.5% slower)

Evaluation

- Dynamic Program Characteristics (Application only)



program	collected calling contexts			PCC	DeltaPath					
	total contexts	max. depth	avg. depth	unique contexts	unique contexts	max. depth	avg. depth	max. UCP	avg. UCP	max. ID
compiler.compiler	92634	15	5.1	141	165	11	2.3	3	1.8	4
compiler.sunflow	63705	12	5.4	156	185	8	2.3	2	1.6	4
compress	3243640985	12	10.0	113	114	2	1.0	2	0.0	31
crypto.aes	14431	9	5.6	194	217	2	1.6	2	1.0	15
crypto.rsa	538625	9	6.0	156	179	2	2.0	2	1.0	9
crypto.signverify	541682	9	6.0	228	242	2	2.0	2	1.0	23
mpegaudio	2489700943	17	13.4	389	427	3	1.0	2	0.0	66
scimark.fft.large	566237360	12	10.0	65	101	3	1.0	2	0.0	4
scimark.lu.large	188838329	10	10.0	53	54	2	1.0	2	0.0	2
scimark.monte_carlo	5033167760	11	10.0	34	35	2	1.0	2	0.0	1
scimark.sor.large	293603875	10	10.0	48	67	3	1.0	2	0.0	2
scimark.sparse.large	252002429	11	10.0	46	47	2	1.0	2	0.0	2
sunflow	2840077292	39	21.8	196612	200452	26	4.4	3	1.0	842711
xml.transform	92333406	55	15.5	24422	24556	25	3.1	3	0.1	66412
xml.validation	12900727	11	9.0	127	141	2	2.0	2	1.0	5

- Average stack depth is 1~4.4 (5.1~21.8 call stack depth)
- PCC collects less unique contexts due to hash collision.
- DeltaPath offers precise decoding compared to PCC.

Conclusion

- DeltaPath provides precise and scalable calling context encoding for procedural and object-oriented programs.
- DeltaPath provides high efficiency similar to PCC with the advantage of precise encoding and decoding.
- DeltaPath deals with dynamic class loading and supports selective encoding.

Feature	PCC	PCCE	DeltaPath
Support both procedural and OO	Y	N	Y
Reliable decoding	N	Y	Y
Scalability	Y*	N	Y

PCC: Probabilistic calling context, PCCE: Precise Calling Context Encoding

* Hash collision may become a problem in very large-scale software.

Thank you