Valence: Variable Length Calling Context Encoding

Tong Zhou, Michael R. Jantz,

Prasad A. Kulkarni, Kshitij A. Doshi, Vivek Sarkar

Common Library Code

Adam

```
void append(List list) {
  lock();
  list.append(…);
  unlock();
}
```

Library code

Common Library Code

Adam

```
void append(List list) {
  lock();
  list.append(…);
  unlock();
}
```

Library code

Conservative

Common Library Code

Adam

```
void append(List list) {
  lock();
  list.append(…);
  unlock();
}
```

Library code

Bob

```
void bob_call() {
  lock();
  append(globalList);
  unlock();
}
```

User code

Common Library Code

Adam

```
void append(List list) {
  lock();
  list.append(…);
  unlock();
}
```

Library code

Bob

```
void bob_call() {
  lock();
  append(globalList);
  unlock();
}
```

User code

Common Library Code

Adam

```
void append(List list) {
   lock();
   list.append(…);
   unlock();
}
```

Library code

Caleb

```
void caleb_call() {
   append(globalList);
}
```

User code

Common Library Code

Adam

```
void append(List list) {
   lock();
   list.append(…);
   unlock();
}
```

Library code

Everything is thread-safe

Caleb

```
void caleb_call() {
   append(globalList);
}
```

User code

Common Library Code

Adam

```
void append(List list) {
   lock();
   list.append(…);
   unlock();
}
```

Library code

Caleb

```
void caleb_call() {
   append(globalList);
}
```

User code

Common Library Code

- If we know the calling context where synchronization is unnecessary, how do we fix it automatically?

After Code Transformation

```
void append(List list) {
  if (call_from_caleb()) {
    lock();
    list.append(…);
    unlock();
  }
  else {
    list.append(…);
  }
}
```

Calling context detection

After Code Transformation

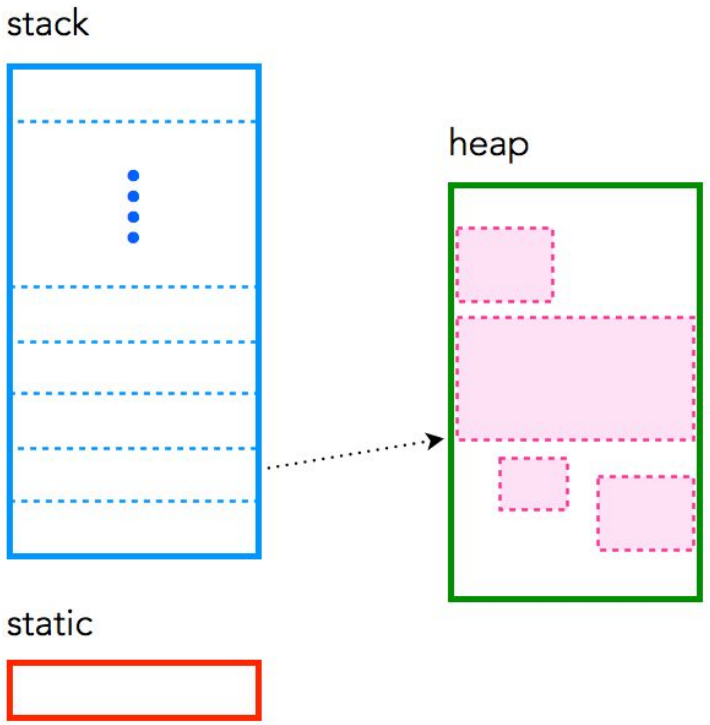
```
void append(List list) {
  if (call_from_caleb()) {
    lock();
    list.append(…);
    unlock();
  }
  else {
    list.append(…);
  }
}
```

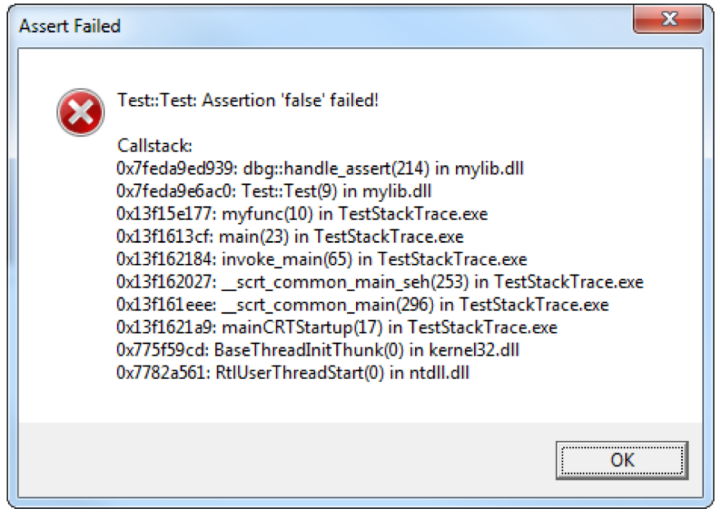
Calling context detection

Synchronization elided

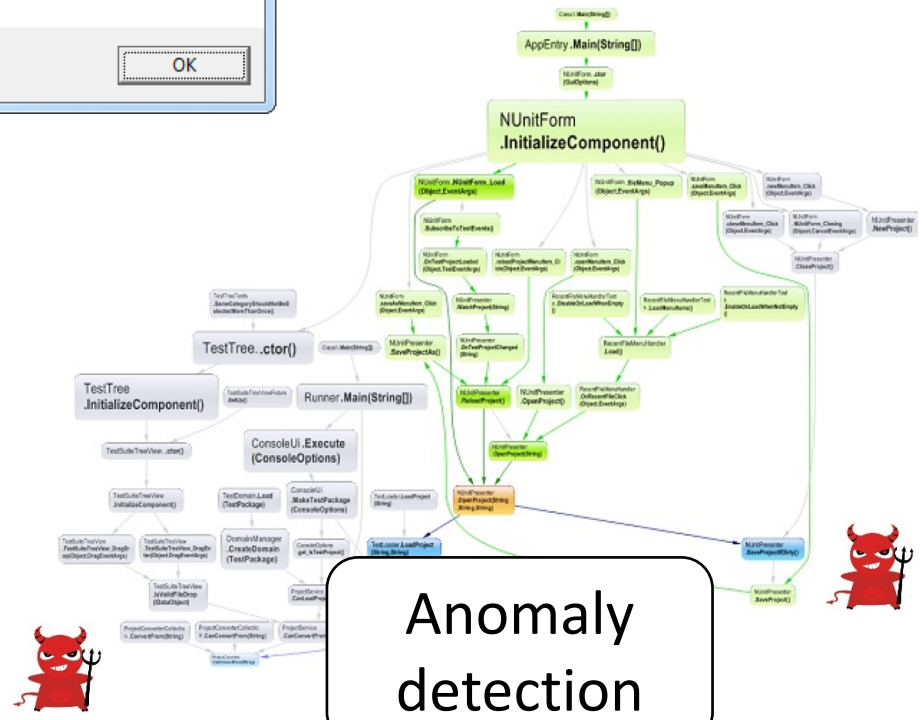
Also Useful For …

Better memory layout





Enhanced Debugging



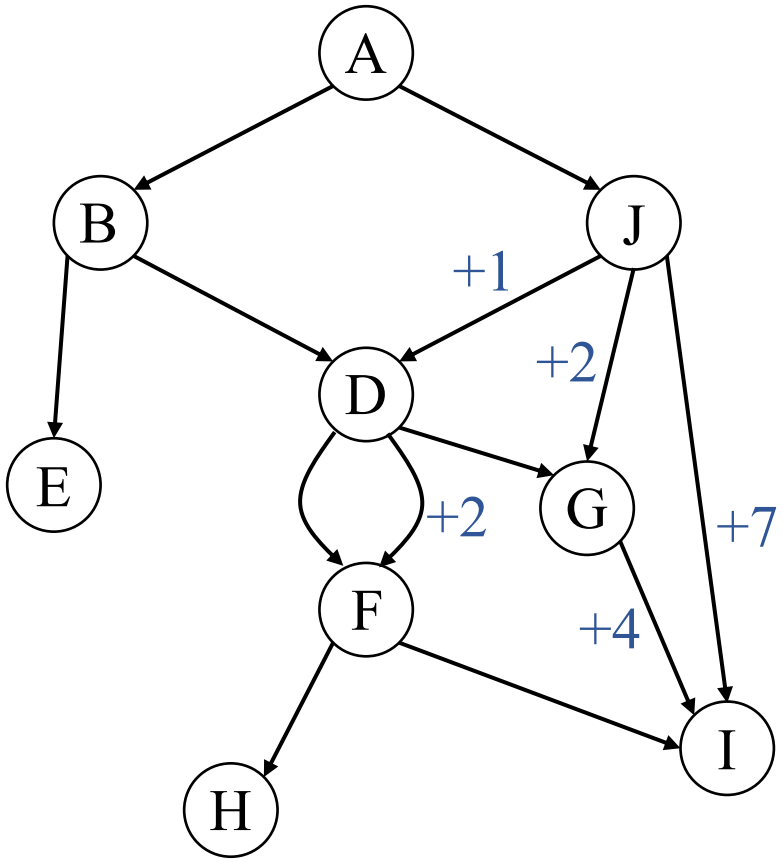
Anomaly detection

However…

- For the current state-of-the-art approach, precise calling context checking could incur:
  - > 8x slowdown when querying at every call site.

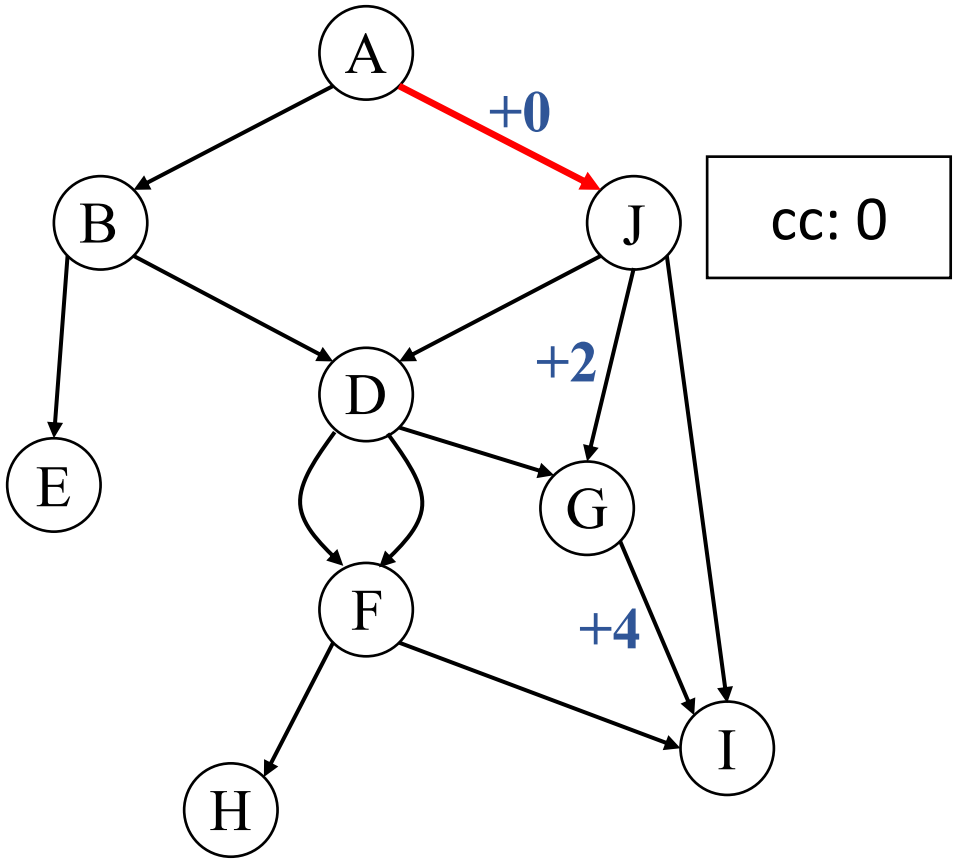
Why this slow? What's the real problem behind it?

Traditional Approach PCCE



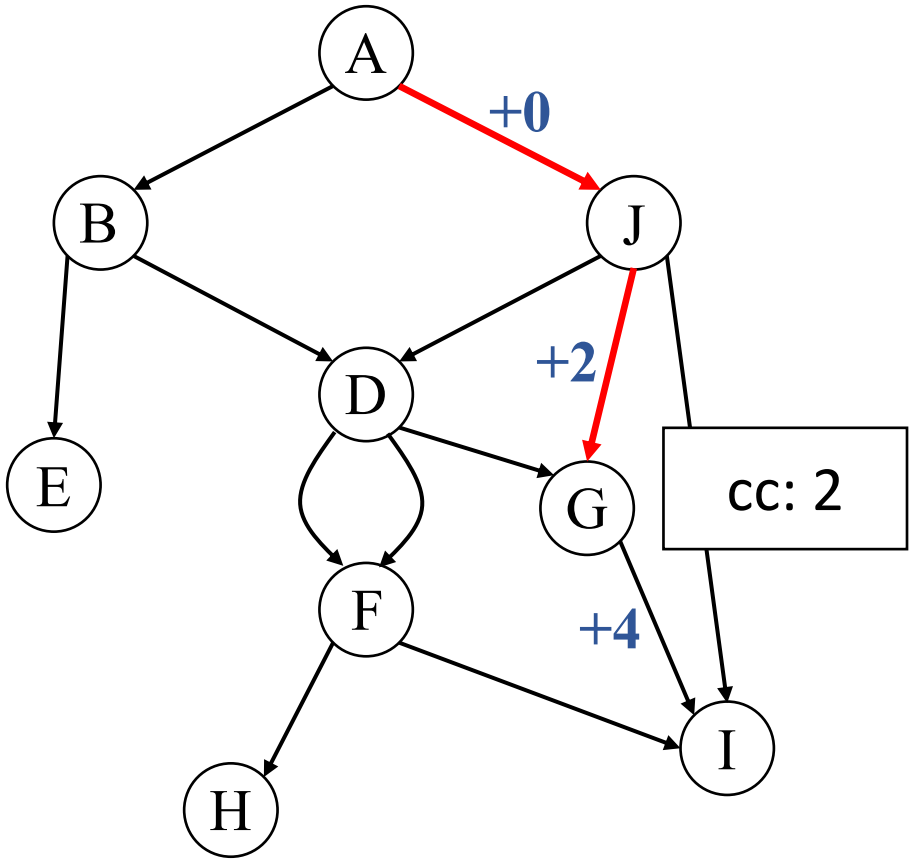
- Use a single integer (*cc*) to encode all contexts
- Assign a unique number to each static context
  - Do integer addition and subtraction on call and call return

Traditional Approach PCCE



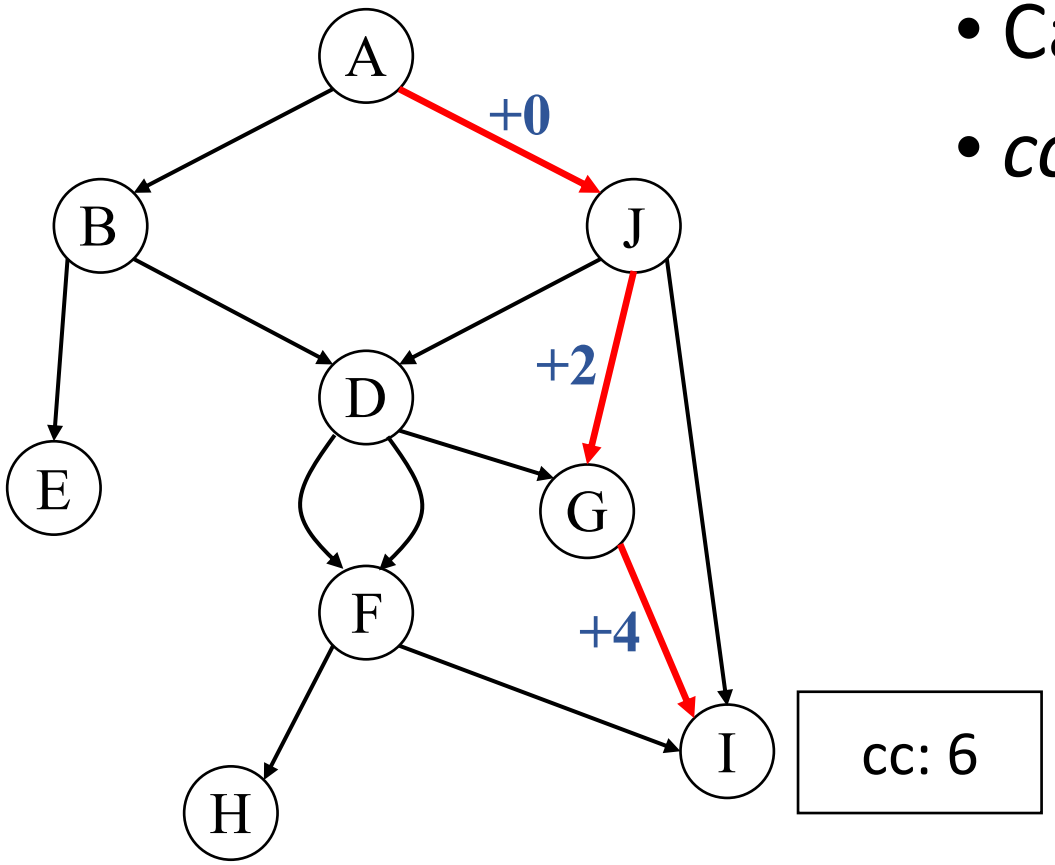
- Calling context: AJ
- *cc*: 0

Traditional Approach PCCE



- Calling context: AJG
- *cc*: 0 + 2 = 2

Traditional Approach PCCE

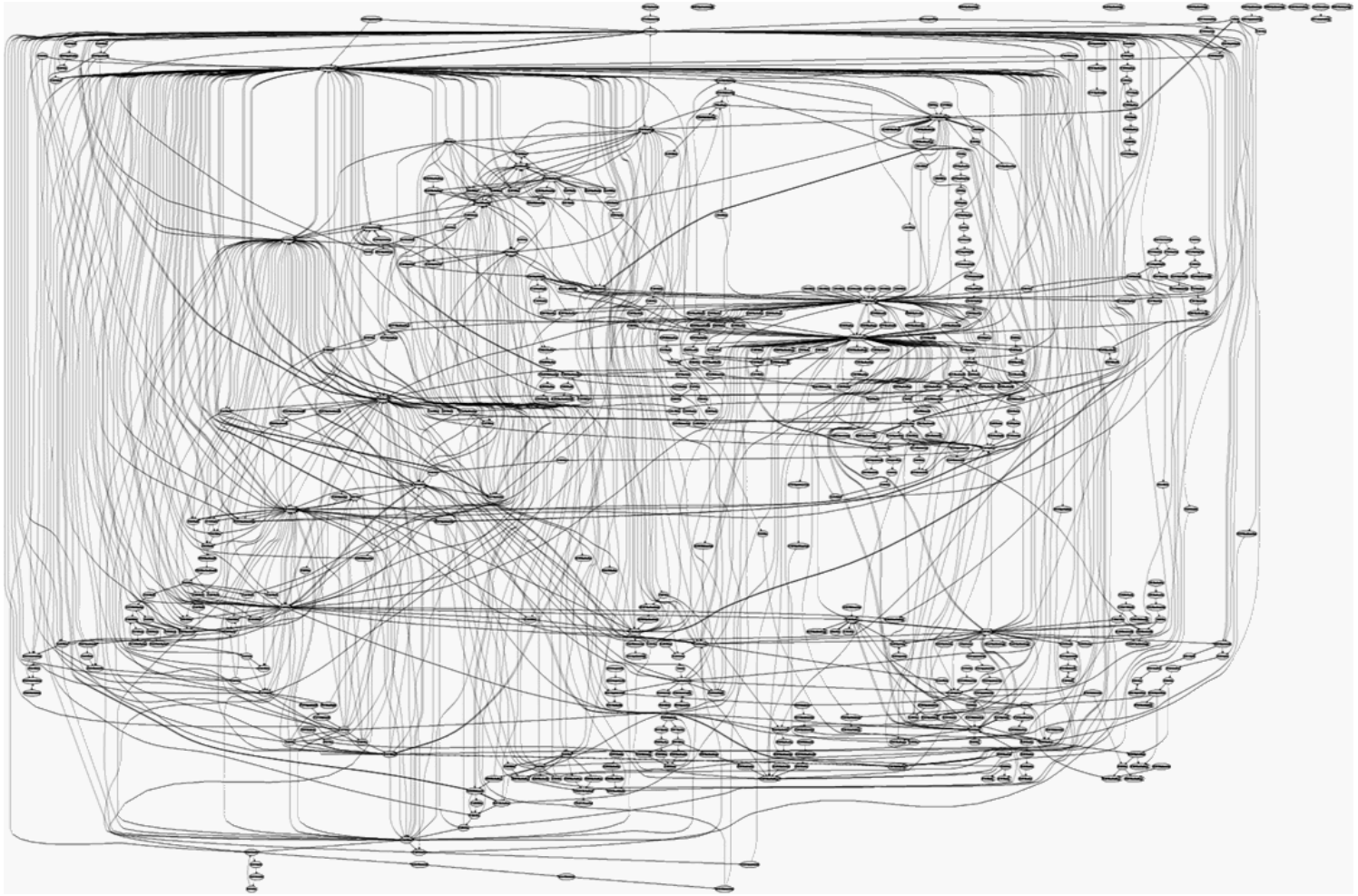


- Calling context: AJGI
- *cc*: 2 + 4 = 6

Problem 1

Massive amount of distinct static calling context for large code base

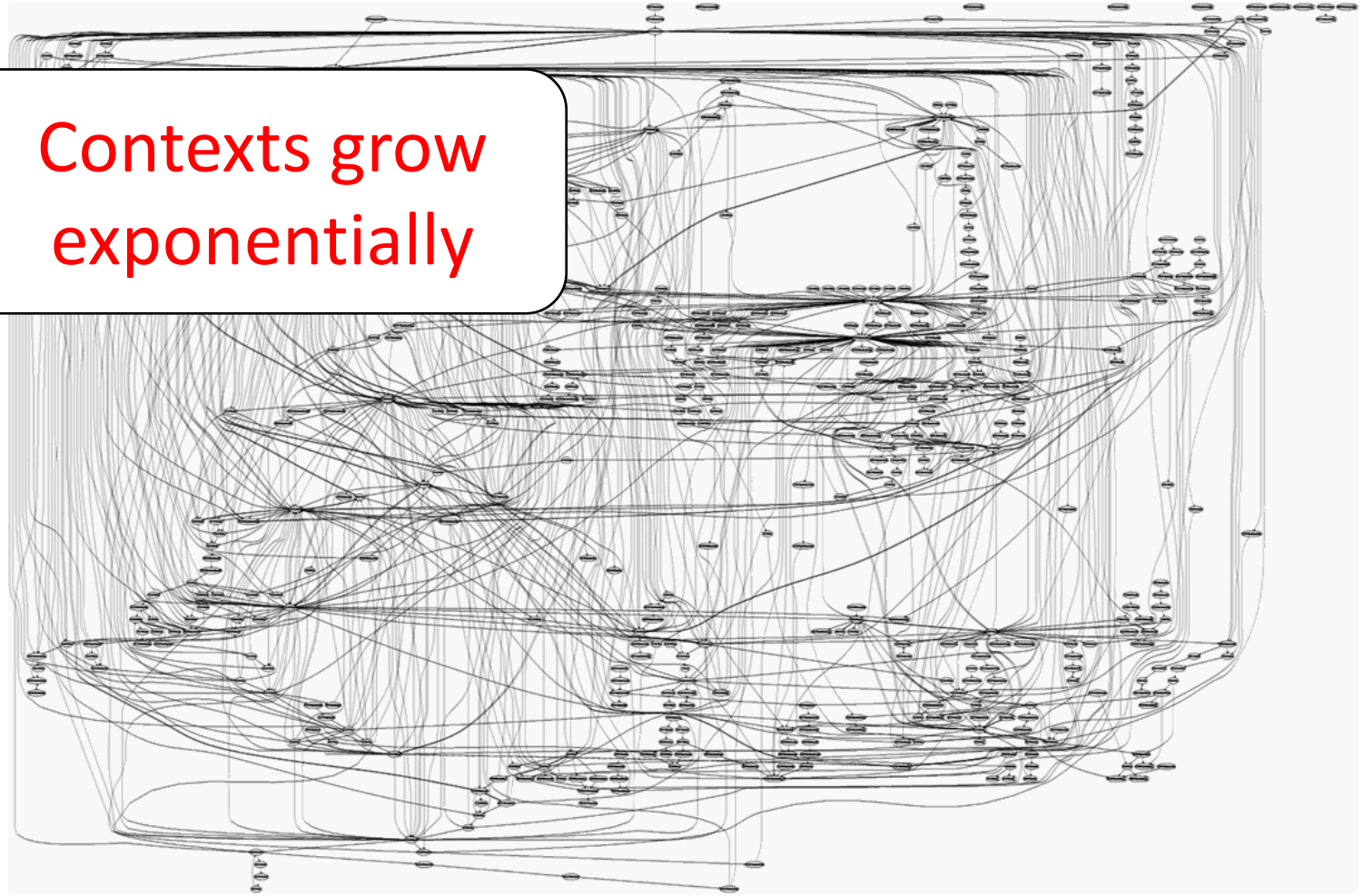
Linux kernel =>



Problem 1

Massive amount of distinct static calling context for large code base

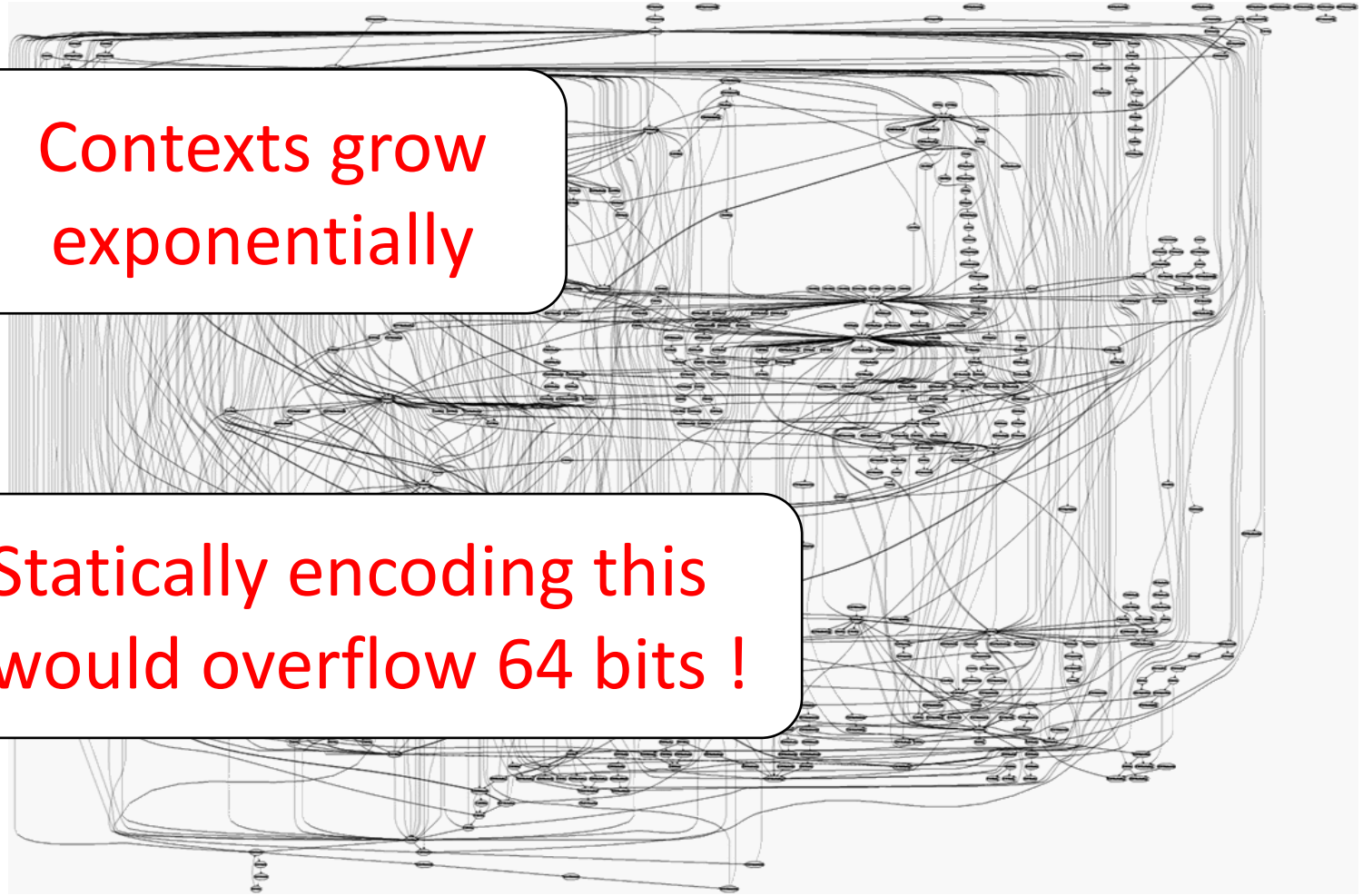
Linux kernel =>



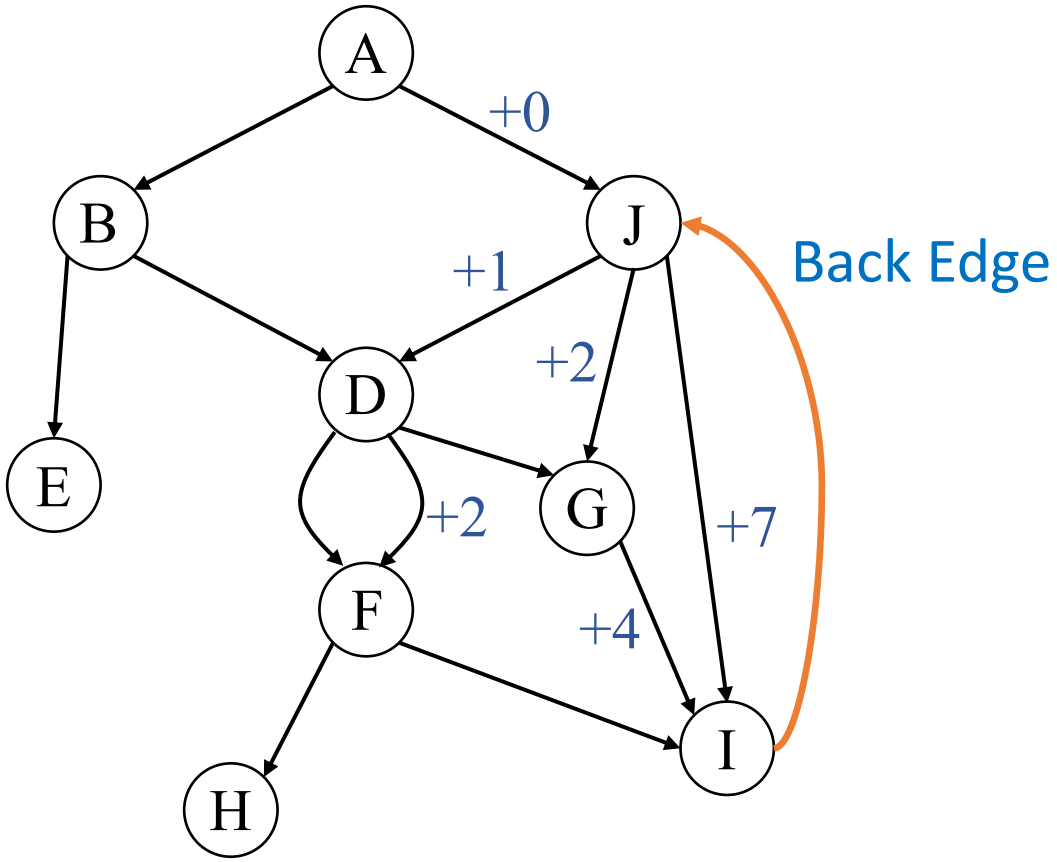
Problem 1

Massive amount of distinct static calling context for large code base

Linux kernel =>

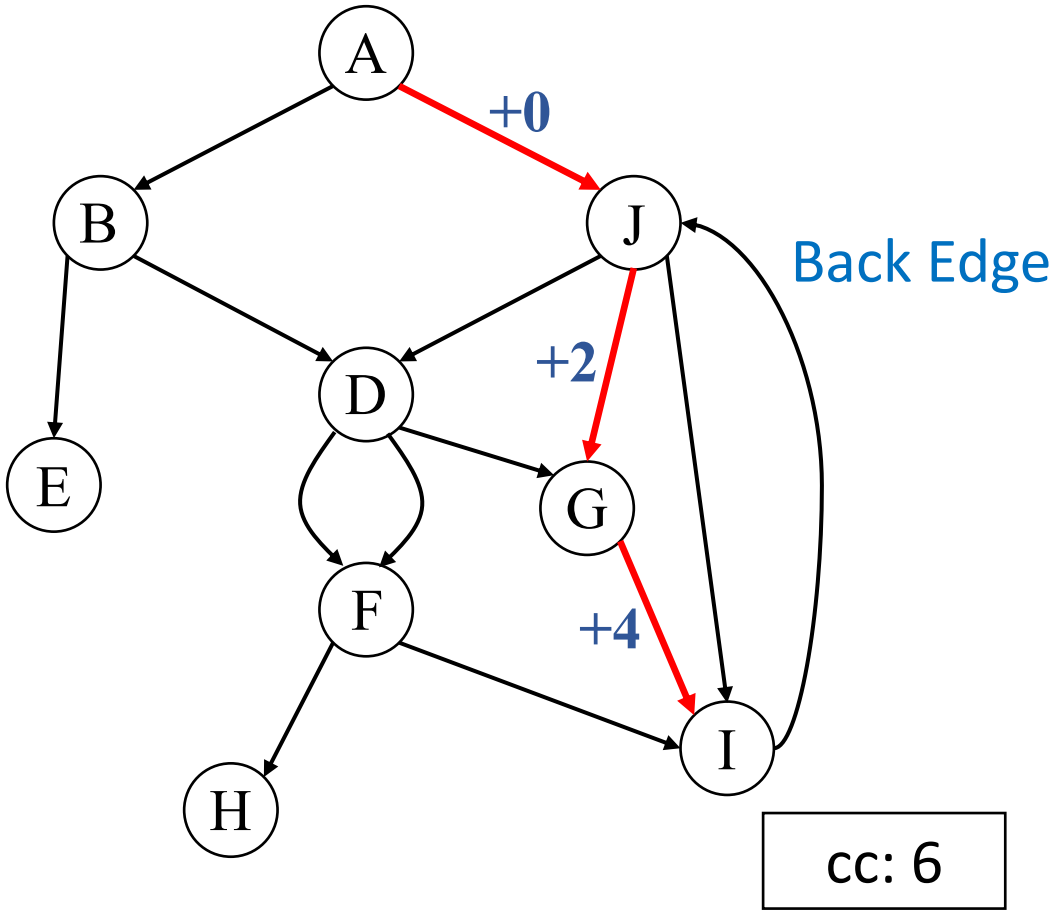


PCCE Deals With Cycles



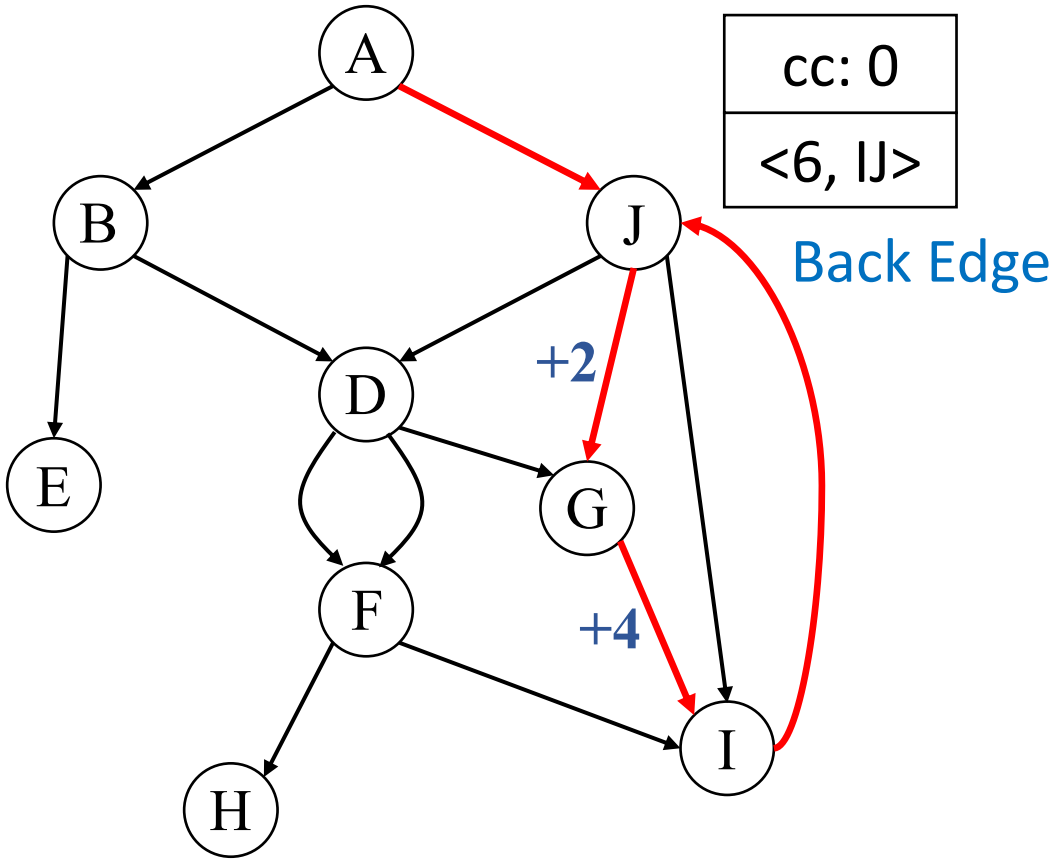
- Push tuple <current *cc*, GJ> onto a stack
- Reset *cc*

Context: AJGI



- Context: AJGI

Context: AJGIJ



- Context: AJGIJ

Problem 2

Need to save the entire acyclic context on each back edge.

- Too much redundant leads to inefficient querying

```
20 ../../../505.mcf_r() [0x40acce]
21 ../../../505.mcf_r() [0x4076fa]
22 ../../../505.mcf_r() [0x4076fa]
23 ../../../505.mcf_r() [0x4076fa]
24 ../../../505.mcf_r() [0x4076fa]
25 ../../../505.mcf_r() [0x4076fa]
26 ../../../505.mcf_r() [0x4076fa]
27 ../../../505.mcf_r() [0x4076fa]
28 ../../../505.mcf_r() [0x4076fa]
29 ../../../505.mcf_r() [0x4076fa]
30 ../../../505.mcf_r() [0x4076fa]
31 ../../../505.mcf_r() [0x4076fa]
32 ../../../505.mcf_r() [0x4076fa]
33 ../../../505.mcf_r() [0x4076fa]
34 ../../../505.mcf_r() [0x4076fa]
35 ../../../505.mcf_r() [0x4076fa]
36 ../../../505.mcf_r() [0x4076fa]
37 ../../../505.mcf_r() [0x4076fa]
38 ../../../505.mcf_r() [0x4076fa]
39 ../../../505.mcf_r() [0x4076fa]
40 ../../../505.mcf_r() [0x4076fa]
41 ../../../505.mcf_r() [0x4076fa]
42 ../../../505.mcf_r() [0x403a4b]
43 ../../../505.mcf_r() [0x400e14]
44 ../../../505.mcf_r() [0x400fc2]
45 /lib64/libc.so.6(__libc_start_main+0xf5) [0x7f6331010b35]
46 ../../../505.mcf_r() [0x400c39]
47
```

Problem 2

Need to save the entire acyclic context on each back edge.

- Too much redundant leads to inefficient querying

```
20 ../../../505.mcf_r() [0x40acce]
21 ../../../505.mcf_r() [0x4076fa]
22 ../../../505.mcf_r() [0x4076fa]
23 ../../../505.mcf_r() [0x4076fa]
24 ../../../505.mcf_r() [0x4076fa]
25 ../../../505.mcf_r() [0x4076fa]
26 ../../../505.mcf_r() [0x
27 ../../../505.mcf_r() [0
28 ../../../505.mcf_r() [0
29 ../../../505.mcf_r() [0
30 ../../../505.mcf_r() [0
31 ../../../505.mcf_r() [0
32 ../../../505.mcf_r() [0
33 ../../../505.mcf_r() [0x
34 ../../../505.mcf_r() [0x4076fa]
35 ../../../505.mcf_r() [0x4076fa]
36 ../../../505.mcf_r() [0x4076fa]
37 ../../../505.mcf_r() [0x4076fa]
38 ../../../505.mcf_r() [0x4076fa]
39 ../../../505.mcf_r() [0x4076fa]
40 ../../../505.mcf_r() [0x4076fa]
41 ../../../505.mcf_r() [0x4076fa]
42 ../../../505.mcf_r() [0x403a4b]
43 ../../../505.mcf_r() [0x400e14]
44 ../../../505.mcf_r() [0x400fc2]
45 /lib64/libc.so.6(__libc_start_main+0xf5) [0x7f6331010b35]
46 ../../../505.mcf_r() [0x400c39]
47
```

Redundancy accumulates

Two Identified Problems

- Current approach
  - > 8x slowdown when querying at every call site.
- Problem 1
  - Unscalable encoding for the massive amount of static calling contexts.
- Problem 2
  - Inefficient encoding for infinite amount of dynamic calling contexts.

Valence Solved Them All

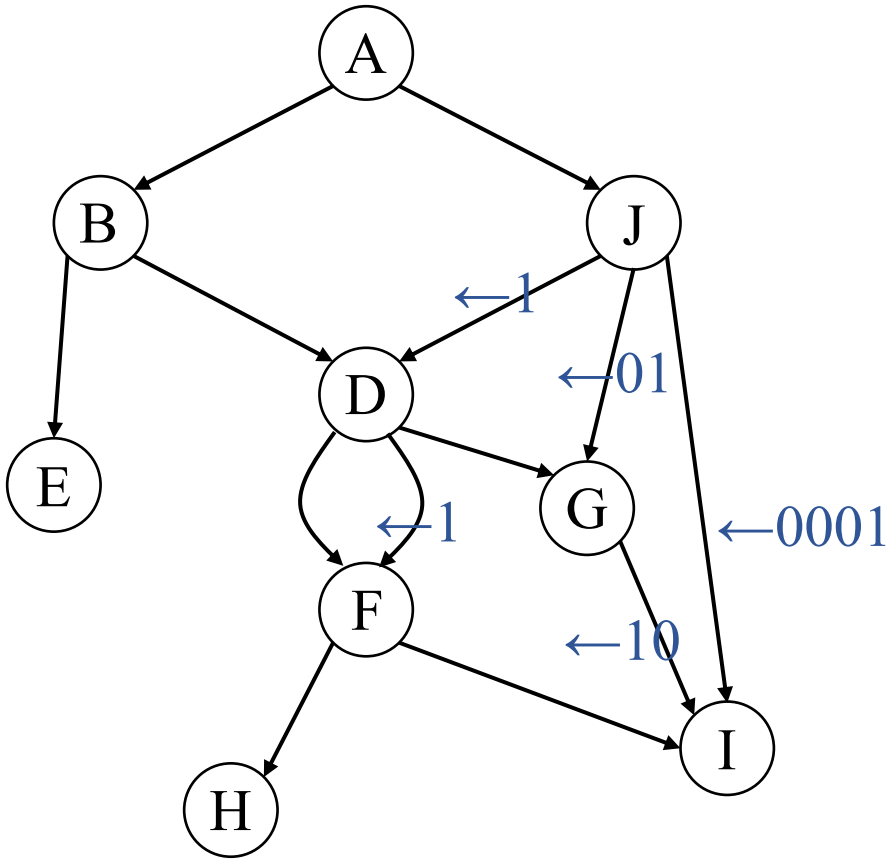
- The most compact scalable precise calling context encoding
- Compared to the current state-of-the-art approach for SPEC CPU2017 benchmarks. On average, Valence achieves
  - > 60% space overhead reduction (from 4.3 to 1.6 64-bit words) for storing calling contexts.
  - > 70% time overhead reduction for querying calling contexts.

Table of Contents

Solution To Scalability Problem

- Encode the context in a logical statically-sized bit vector.
  - Naturally scalable
- Instrumentations
  - Before the call: append a value to the bit vector
  - After the call: pop out the value

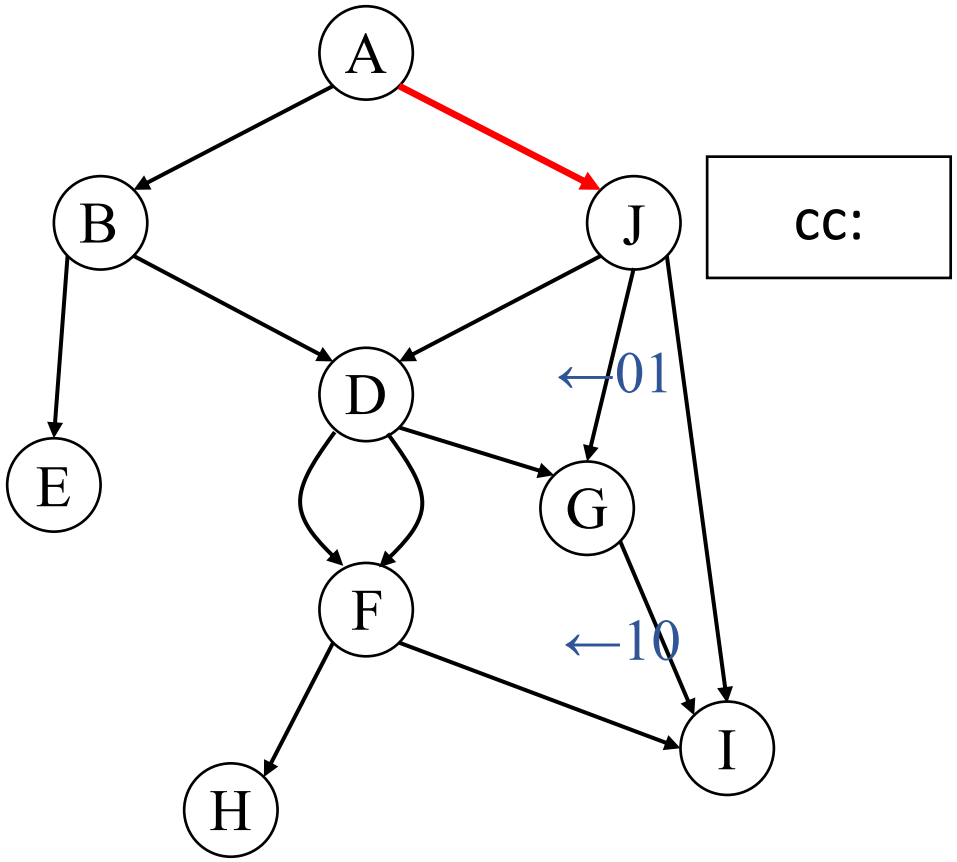
Valence Acyclic Encoding



← : append a binary number

- Use a static bit vector to encode all contexts
- Assign a unique bit pattern to each static context
  - Do bits appending and poping on call and call return (logical)

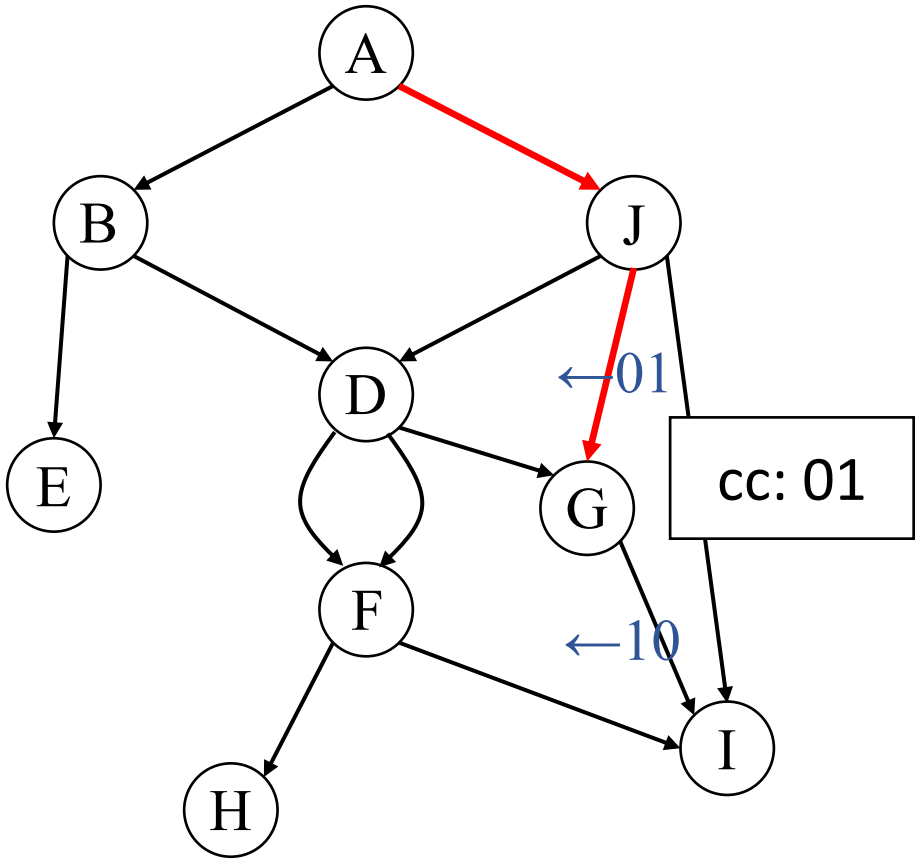
Valence Acyclic Encoding



- Calling context: AJ
- *cc*: (nil)

← : append a binary number

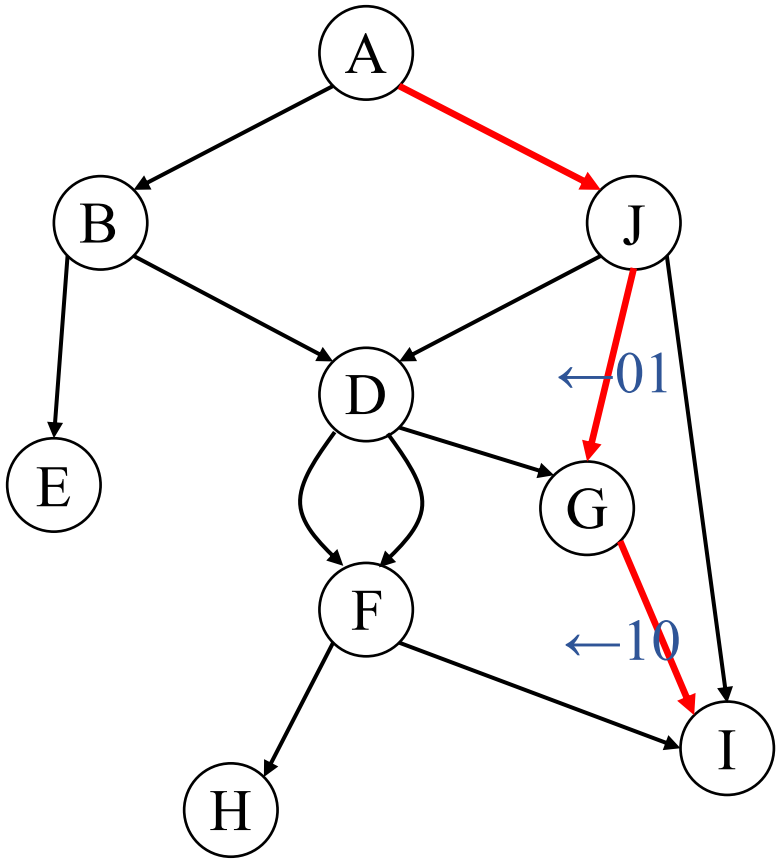
Valence Acyclic Encoding



- Calling context: AJG
- *cc*: 01

← : append a binary number

Valence Acyclic Encoding



cc: 01 10

- Calling context: AJGI
- *cc*: 0110

← : append a binary number

Details Are In The Paper

- How do we statically determine what range in the bit vector to update at each call site?
- How to ensure each bit pattern is unique?
- Check out the algorithms in the paper.

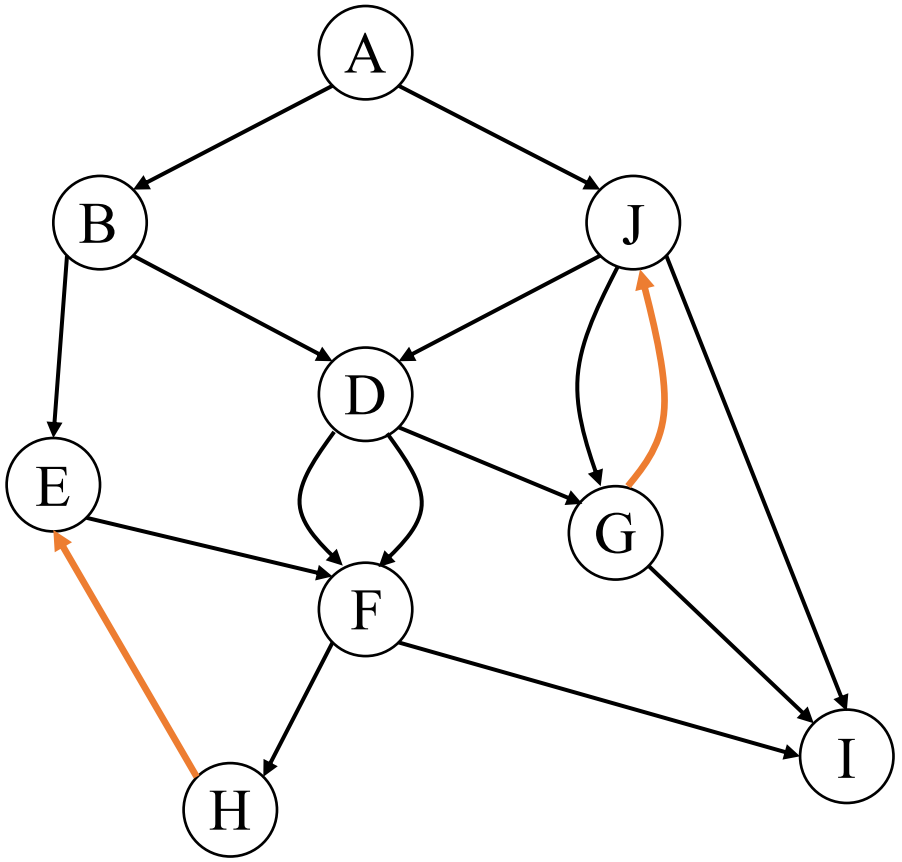
Table of Contents

Solution To Cycle Problem

- Goal: to reduce encoding sizes and thus improve performance

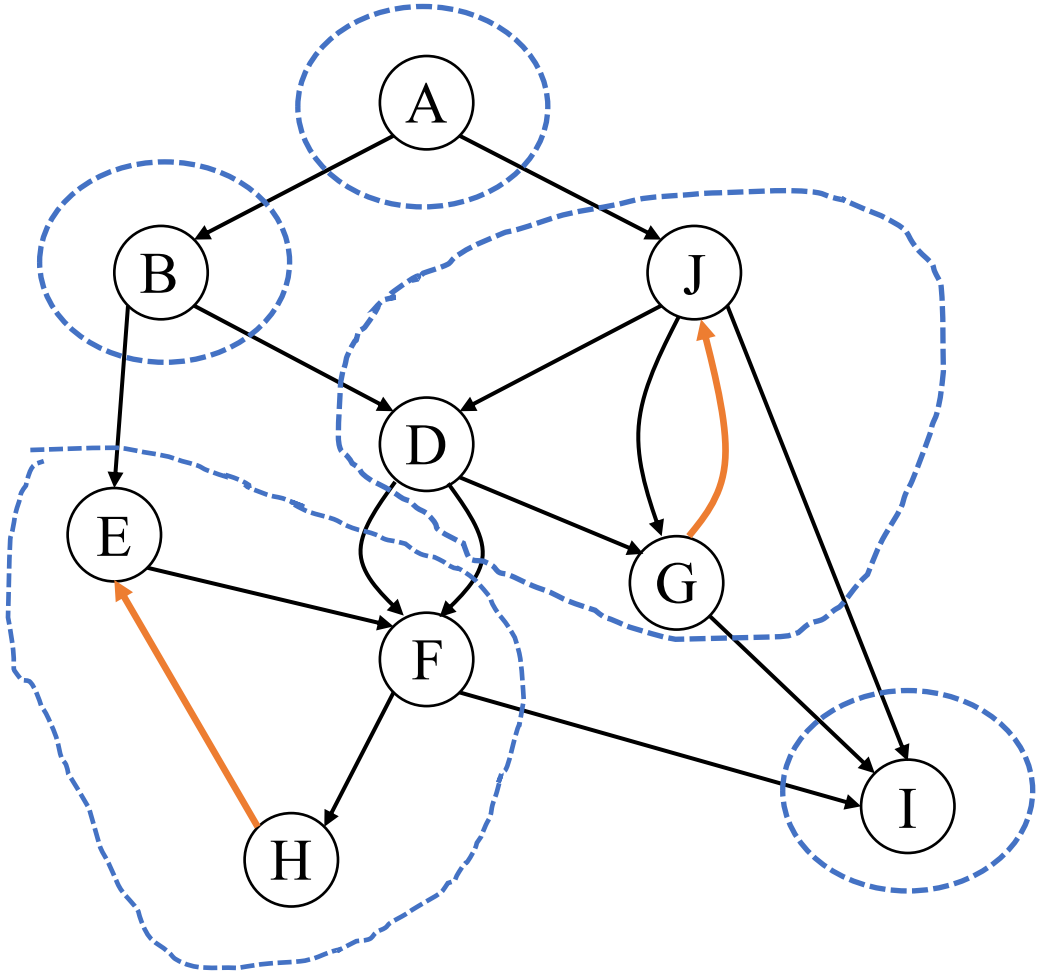
- A different way to define cycle edges.
- First calculate the strongly-connected components (SCCs) of the call graph.

A Call Graph With Cycles



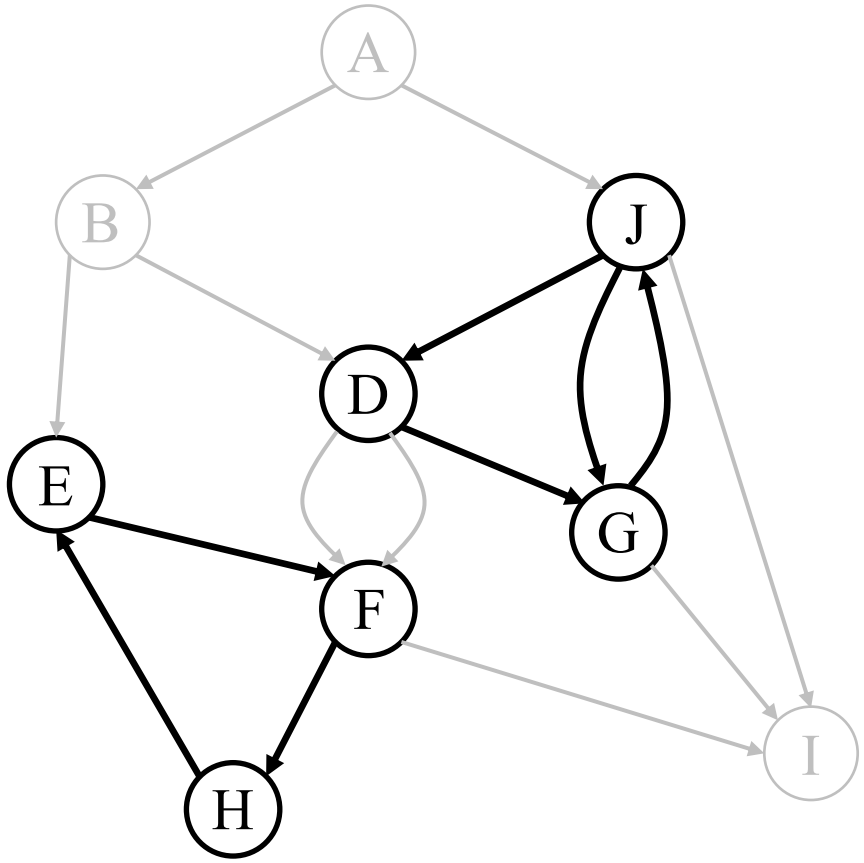
- Back edges: HE, GJ

A Call Graph With Cycles



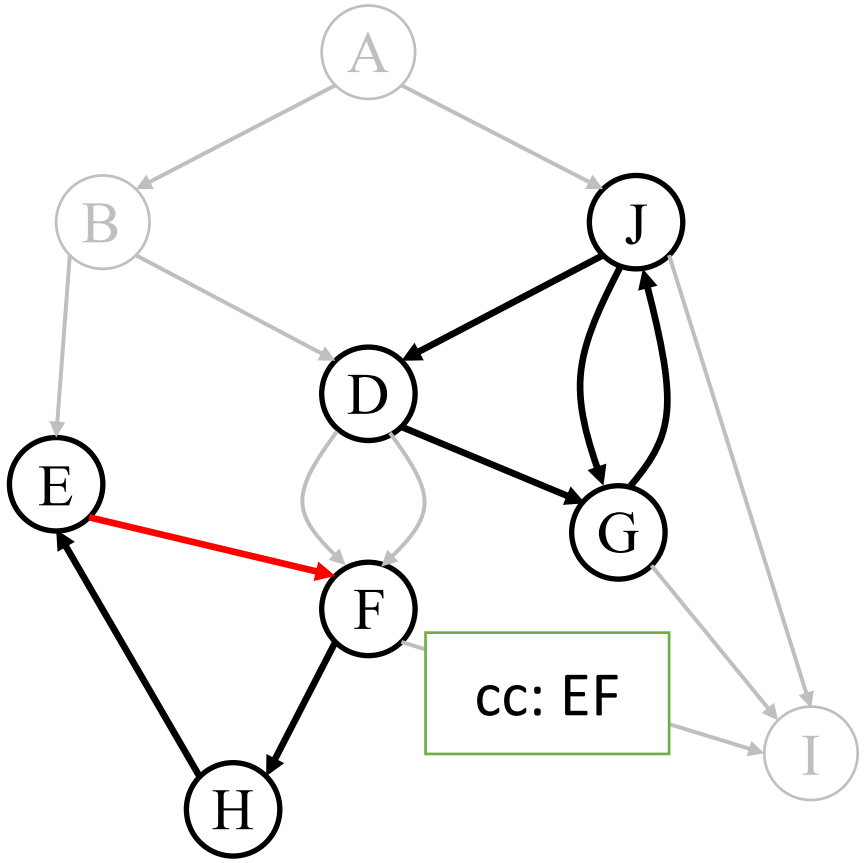
- Five SCCs

Cycle Edges



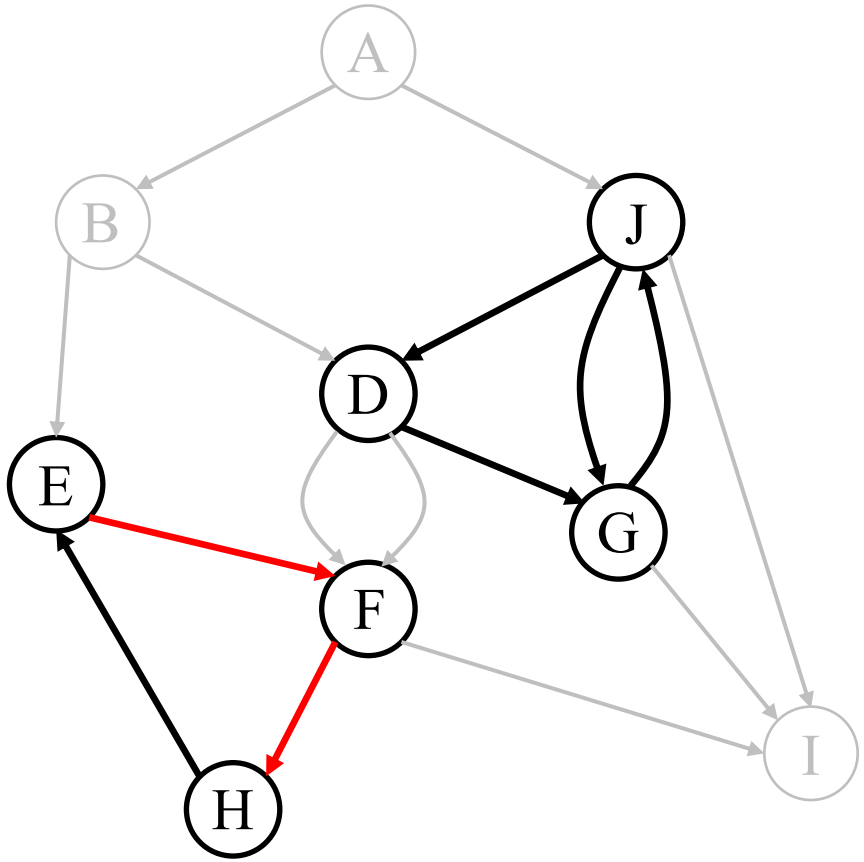
- SCC EFH and DGJ have edges inside
  - Cycle edges
- Only need to store these edges for cycle encoding
  - Store in a dynamic bit vector

Cycle Encoding



- Cyclic context: EF

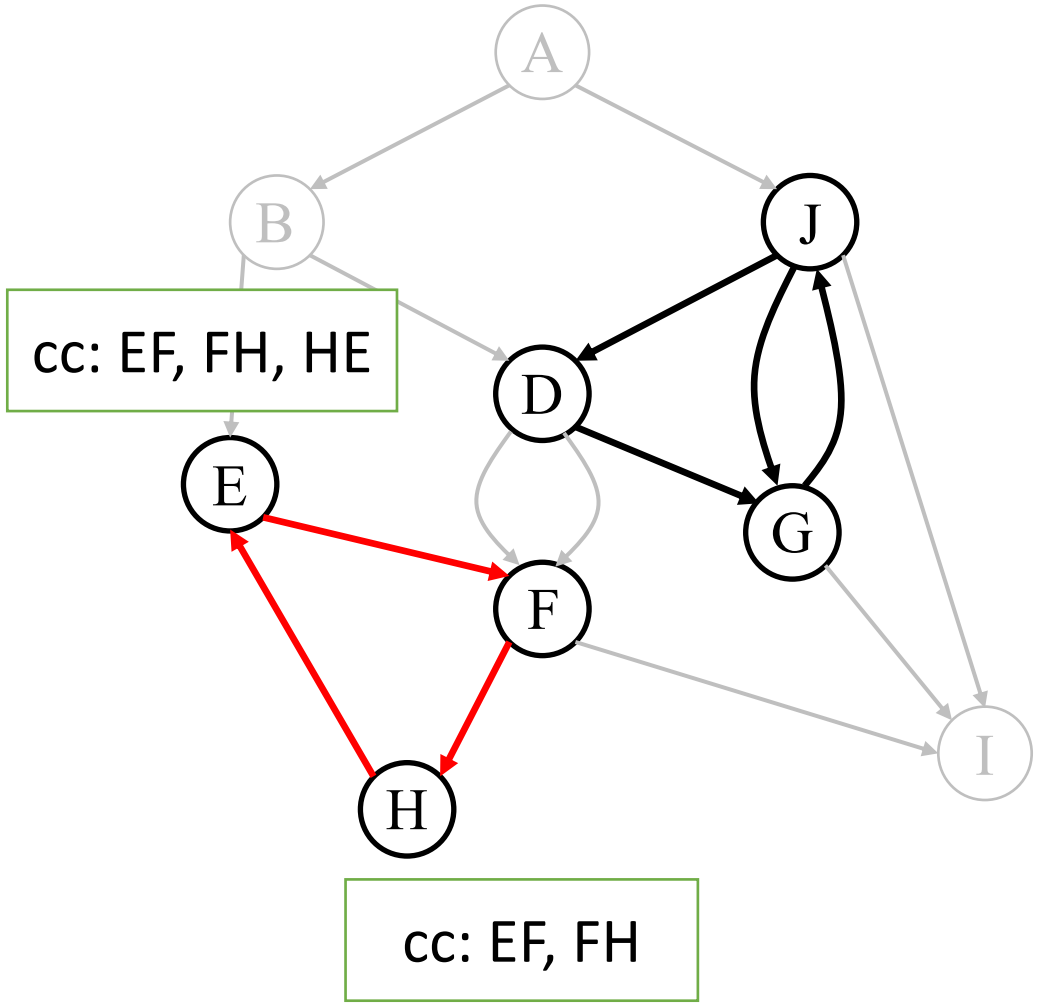
Cycle Encoding



cc: EF, FH

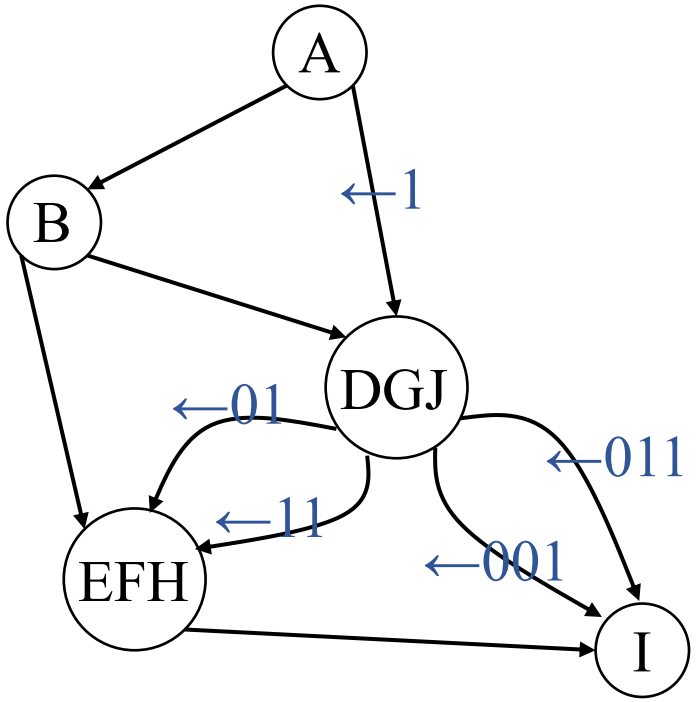
- Cyclic context: EFH

Cycle Encoding



- Cyclic context: EFHE

Instrument Acyclic Edges



- The SCCs form an acyclic graph
- Use the previous acyclic encoding scheme

Takeaways

- Encode call graphs without cycles
  - Use a statically-sized bit vector
  - Scale efficiently
- Encode cycles in the call graphs
  - Use a separate dynamically-sized bit vector
  - Space efficient

Table of Contents

We Want To Know…

- Acyclic call graph
  - how much more space Valence requires than PCCE?
- Cyclic call graph
  - how much more compact Valence is?
    - A more compact encoding makes querying more efficient (less memory traffic)
- Valence v.s. PCCE
  - Instrumentation overhead
  - Detection overhead

Evaluation

- Configurations
  - Each encoding strategy (PCCE, Valence) is implemented as a LLVM pass.
- Hardware
  - 3.30GHz Intel CPU and 16G DRAM.
- Benchmark
  - SPEC CPU2017 C/C++ Benchmark suite.

Table of Contents

Benchmark Static Characteristics

Benchmark	Call Graph Statistics				
	Nodes	Edges	SCCs	Complex SCCs	Cyclic Edges
gcc	19,011	131,388	17,182	459	28,330
mcf	32	126	32	1	1
cactuBSSN	1,048	22,820	1,040	12	67
namd	61	553	61	0	0
parest	1,315	15,080	1,234	46	200
povray	519	8,258	377	57	2,380
lbm	17	27	17	0	0
xalancbmk	4,055	22,848	3,700	181	1,566
x264	367	2,318	366	1	2
deepsjeng	87	573	87	3	12
imagick	915	18,864	807	40	524
leela	204	1,013	202	15	23
nab	79	730	79	10	25
xz	149	359	140	3	18

Benchmark Static Characteristics

Benchmark	Call Graph Statistics				
	Nodes	Edges	SCCs	Complex SCCs	Cyclic Edges
gcc	19,011	131,388	17,182	459	28,330
mcf	32	126	32	1	1
cactuBSSN	1,048	22,820	1,040	12	67
namd	61	553	61	0	0
parest	1,315	15,080	1,234	46	200
povray	519	8,258	377	57	2,380
lbm	17	27	17	0	0
xalancbmk	4,055	22,848	3,700	181	1,566
x264	367	2,318	366	1	2
deepsjeng	87	573	87	3	12
imagick	915	18,864	807	40	524
leela	204	1,013	202	15	23
nab	79	730	79	10	25
xz	149	359	140	3	18

Cycle problem

Benchmark Static Characteristics

Benchmark	Call Graph Statistics				
	Nodes	Edges	SCCs	Complex SCCs	Cyclic Edges
gcc	19,011	131,388	17,182	459	28,330
mcf	32	126	32	1	1
cactuBSSN	1,048	22,820	1,040	12	67
namd	61	553	61	0	0
parest	1,315	15,080	1,234	46	200
povray	519	8,258	377	57	2,380
lbm	17	27	17	0	0
xalancbmk	4,055	22,848	3,700	181	1,566
x264	367	2,318	366	1	2
deepsjeng	87	573	87	3	12
imagick	915	18,864	807	40	524
leela	204	1,013	202	15	23
nab	79	730	79	10	25
xz	149	359	140	3	18

At most 15 bits to encode a cycle edge

Cyclic Encoding Cost Estimation

Benchmark	PCCE (bits)	Valence (bits)	Valence/PCCE
gcc	1E+06	424950	34%
mcf	6	1	17%
cactusBSSN	1802	469	26%
parest	2656	1600	60%
povray	62913	28560	45%
xalancbmk	48932	17226	35%
x264	18	2	11%
deepsjeng	228	48	21%
imagick	18144	5240	29%
leela	378	115	30%
nab	450	125	28%
xz	126	90	71%

Geomean:
49%

Detection Overhead

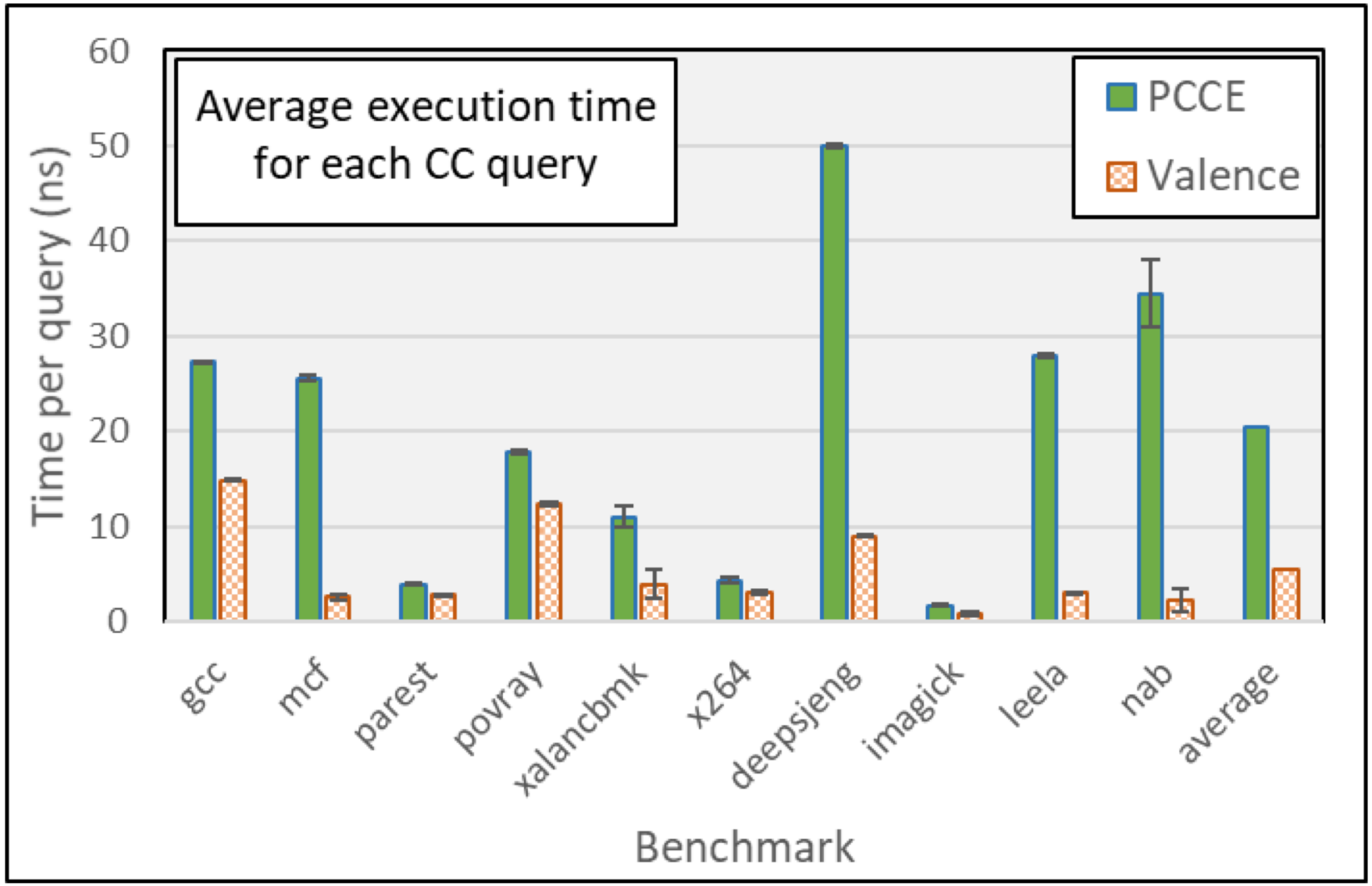


Table of Contents

Benchmark Static Characteristics

Benchmark	Acyclic Encoding Bits	
	PCCE	Valence
gcc	214	148
mcf	6	7
cactuBSSN	28	55
namd	10	13
parest	25	42
povray	57	57
lbm	2	2
xalancbmk	42	87
x264	18	27
deepsjeng	15	21
imagick	64	113
leela	13	17
nab	13	16
xz	10	18

Benchmark Static Characteristics

Benchmark	Acyclic Encoding Bits	
	PCCE	
gcc	214	
mcf	6	
cactuBSSN	28	
namd	10	13
parest	25	42
povray	57	57
lbm	2	2
xalancbmk	42	87
x264	18	27
deepsjeng	15	21
imagick	64	113
leela	13	17
nab	13	16
xz	10	18

PCCE Cannot operate on 214-bit integer efficiently

Benchma

Average Acyclic Words

PCCE 1.21

Valence 1.29

Benchmark	Acyclic Encoding Bits	
	PCCE	Valence
gcc	214	148
mcf	6	7
cactuBSSN	28	55
namd	10	13
parest	25	42
povray	57	57
lbm	2	2
xalancbmk	42	87
x264	18	27
deepsjeng	15	21
imagick	64	113
leela	13	17
nab	13	16
xz	10	18

Benchma

Average Acyclic Words

PCCE 1.21

Valence 1.29

Benchmark	Acyclic Encoding Bits	
	PCCE	Valence
gcc	214	148
mcf	6	7
cactuBSSN	28	55
namd	10	13
parest	25	42
povray	57	57
lbm	2	2
xalancbmk	42	87
x264	18	27
deepsjeng	15	21
imagick	64	113
leela	13	17
nab	13	16
xz	10	18

Low cost for scalability!

Instrumentation Overhead

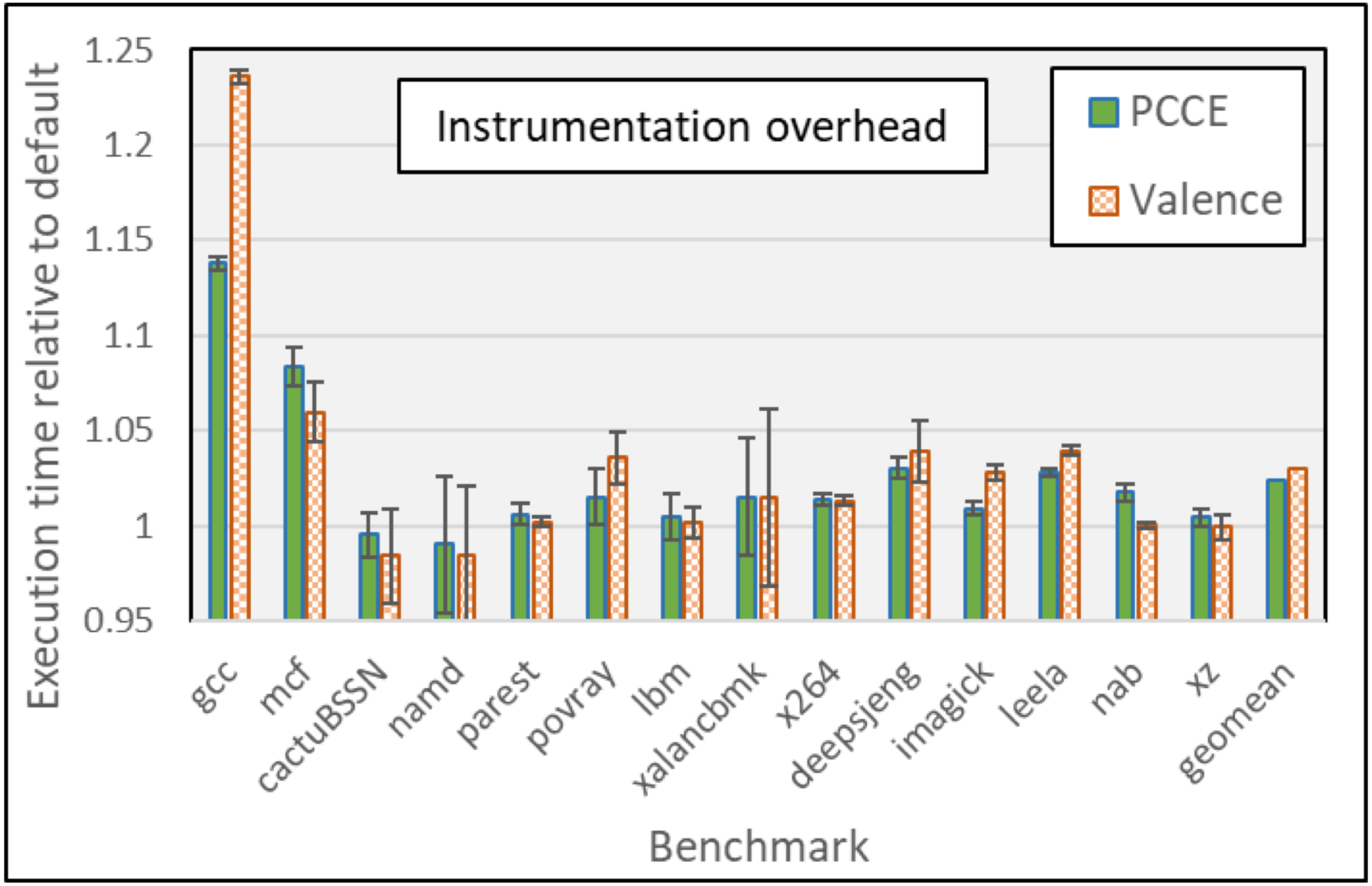


Table of Contents

Conclusion & Future

- We presented Valence: a precise context encoding scheme that is both scalable and low overhead to query.
- Overall, our approach reduces the length of calling context encoding from 4.3 words to 1.6 words on average (> 60% reduction), thereby improving the efficiency of applications that frequently store or query calling contexts.
- See how Valence can enhance some program analysis and software engineering fields.

Thank you! Questions?

Contact info: tz@gatech.edu

Back up slides

DeltaPath: A Scalable Version of PCCE

- Encode context using a list of <cc, anchor node> pairs
- Tuple list operation is difficult to implement efficiently
- Still inherits PCCE's inefficient cycle encoding

DeltaPath: A Scalable Version of PCCE

- Divide the call graph into sub-graphs
  - Each subgraph is encodable with PCCE
- Introduce the notion of “anchor node” as entry point for each subgraph

DeltaPath: A Scalable Version of PCCE