Faster Mutation Analysis via Equivalence Modulo States

Bo Wang, **Yingfei Xiong**, Yangqingwei Shi,
Lu Zhang, Dan Hao

Peking University

July 12, 2017

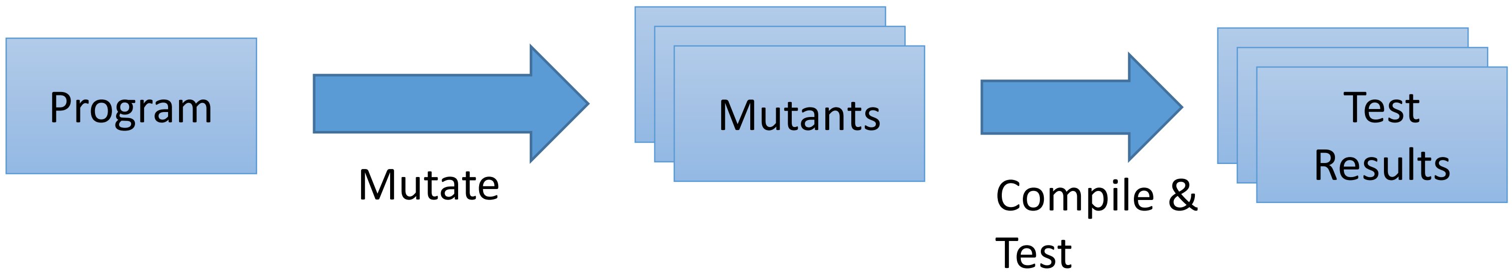
高可信软件技术
教育部重点实验室
Key Lab of High Confidence Software Technologies
(Peking University), Ministry of Education

Mutation Analysis

- Mutation analysis is a fundamental software analysis technique
  - **Mutation Testing** [DeMillo & Lipton, 1970]
  - Mutation-Based **Test Geneartion** [Fraser & Zeller, 2012]
  - Determining **Mutant Utility** [Just et al., 2017]
  - Mutation-based **Fault Localization** [Papadakis & Traon, 2012]
  - Generate-Validate **Program Repair** [Weimer et al., 2013]
  - Testing **Software Product Lines** [Devroey et al., 2014]

Scalability: A Key Limiting Issue

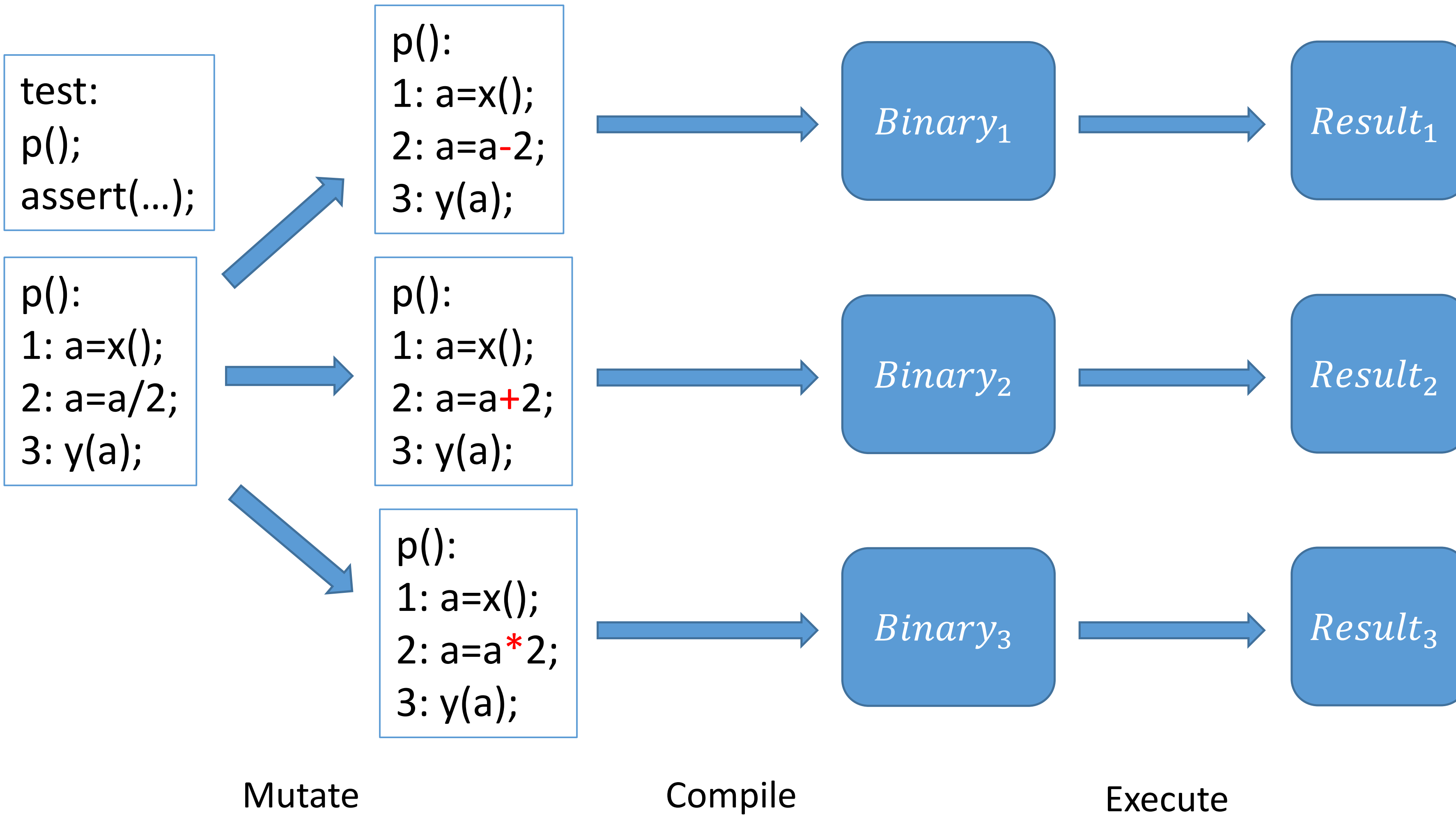
- The testing time of a single program is amplified N times
  - N is the number of mutants
  - N can be usually large
  - N is related to the size of the program
- Plain mutation analysis scales to only programs less than 10k lines of code



Redundant Computations

- Many computation steps in mutation analysis are equivalent
- Reusing them could possibly enhance scalability

Example


test:
p();
assert(...);
p():
1: a=x();
2: a=a/2;
3: y(a);
p():
1: a=x();
2: a=a-2;
3: y(a);
p():
1: a=x();
2: a=a+2;
3: y(a);
p():
1: a=x();
2: a=a*2;
3: y(a);
$Binary_1$
$Binary_2$
$Binary_3$
$Result_1$
$Result_2$
$Result_3$
Mutate
Compile
Execute

Existing work 1:
Mutation Schemata [Untch, Offutt, Harrold, 1993]

```
p():
1: a=x();
2: a=a-2;
3: y(a);
x():
…
y():
…
```

```
p():
1: a=x();
2: a=a+2;
3: y(a);
x():
…
y():
…
```

```
p():
1: a=x();
2: a=a*2;
3: y(a);
x():
…
y():
…
```

- Procedures x() and y() are the same in the three mutants, but they are compile three times
- Redundancy in Compilation

Existing work 1:
Mutation Schemata [Untch, Offutt, Harrold, 1993]

```
p():
1: a=x();
2: a=a-2;
3: y(a);
```

```
p():
1: a=x();
2: a=a+2;
3: y(a);
```

```
p():
1: a=x();
2: a=a*2;
3: y(a);
```

```
p():
1: a=x();
2: if(mut==1) a=a-2
   else if (mut==2) a=a+2
   else a=a*2;
3: y(a);
```

- Generate one big program that compiles once
- Mutants are selected dynamically through input parameters

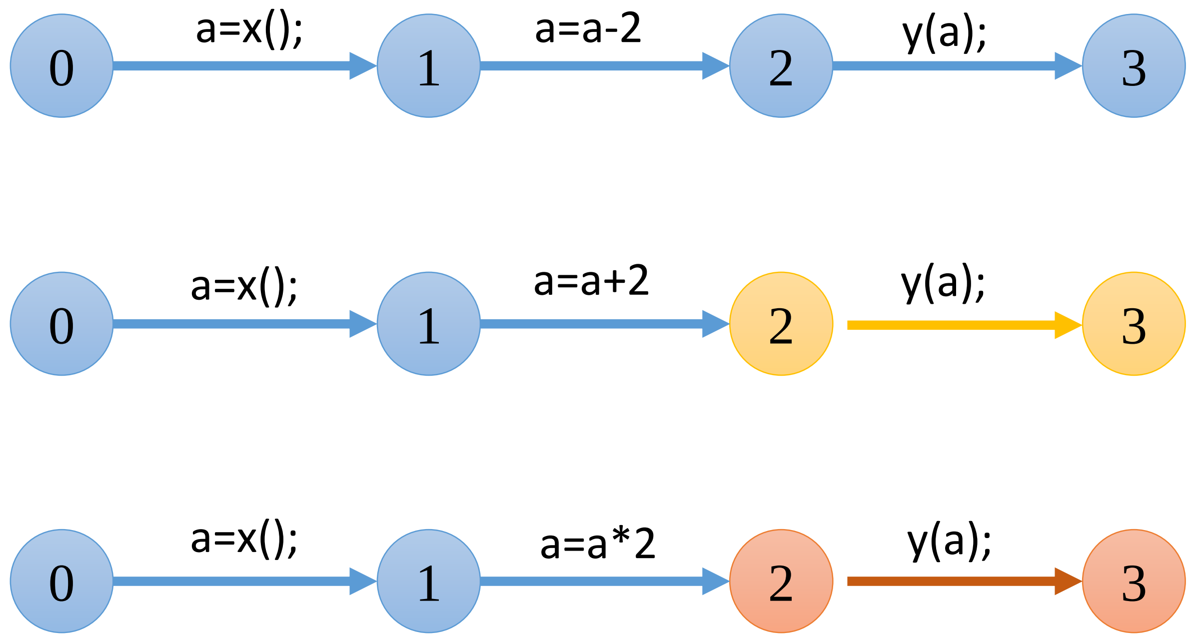
Existing work 2: Split-Stream Execution

[King, Offutt, 1991][Tokumoto et al., 2016][Gopinath, Jensen, Groce, 2016]

```
1: a=x();
2: a=a-2;
3: y(a);
```

```
1: a=x();
2: a=a+2;
3: y(a);
```

```
1: a=x();
2: a=a*2;
3: y(a);
```



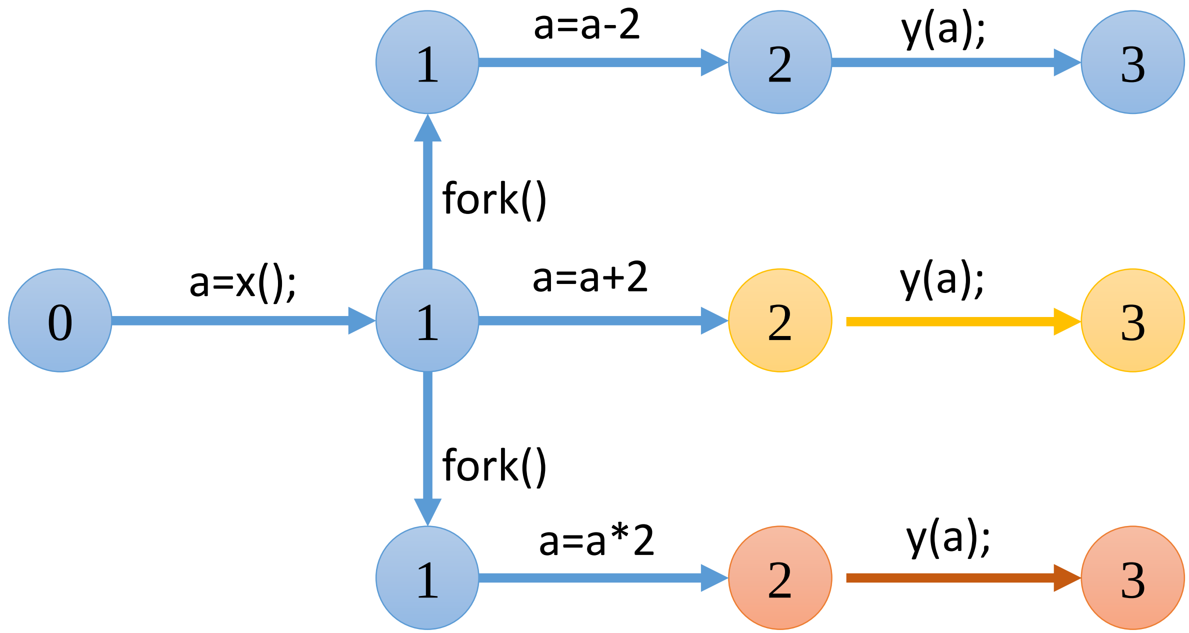
- The computations before the first mutated statement are redundant

Existing work 2: Split-Stream Execution

```
1: a=x();
2: a=a-2;
3: y(a);
```

```
1: a=x();
2: a=a+2;
3: y(a);
```

```
1: a=x();
2: a=a*2;
3: y(a);
```



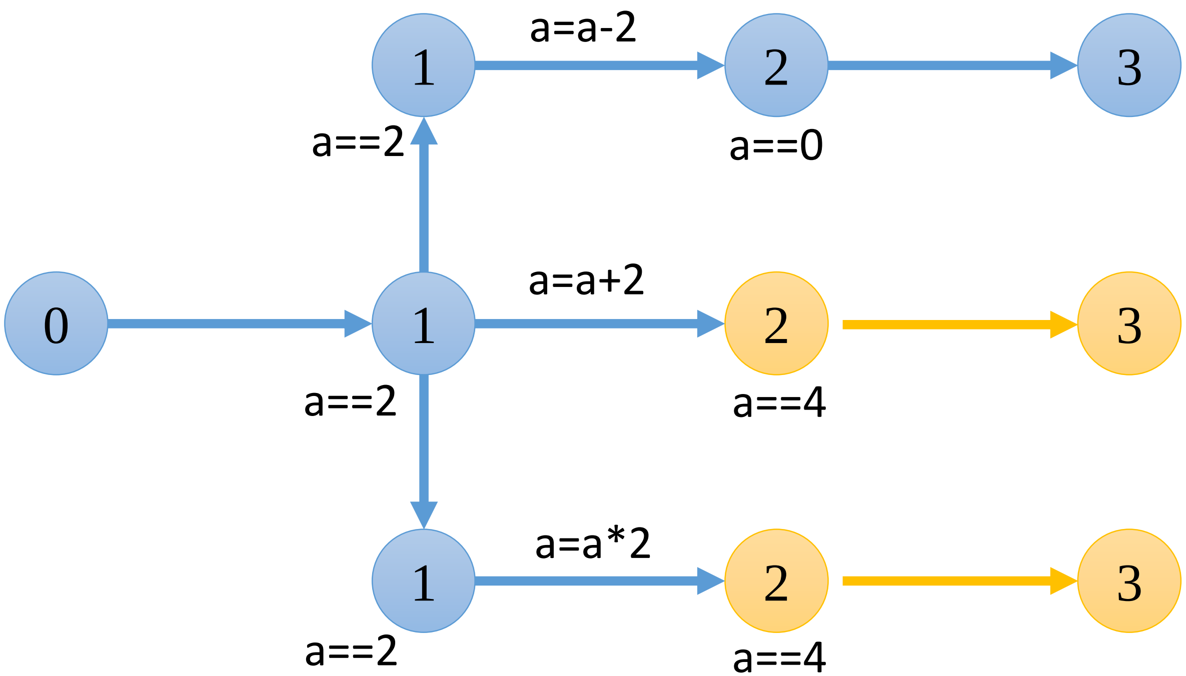
- Start with one process
- Fork processes when mutated statements are encountered

Redundancy After the First Mutated Statement

```
1: a=x();
2: a=a-2;
3: y(a);
```

```
1: a=x();
2: a=a+2;
3: y(a);
```

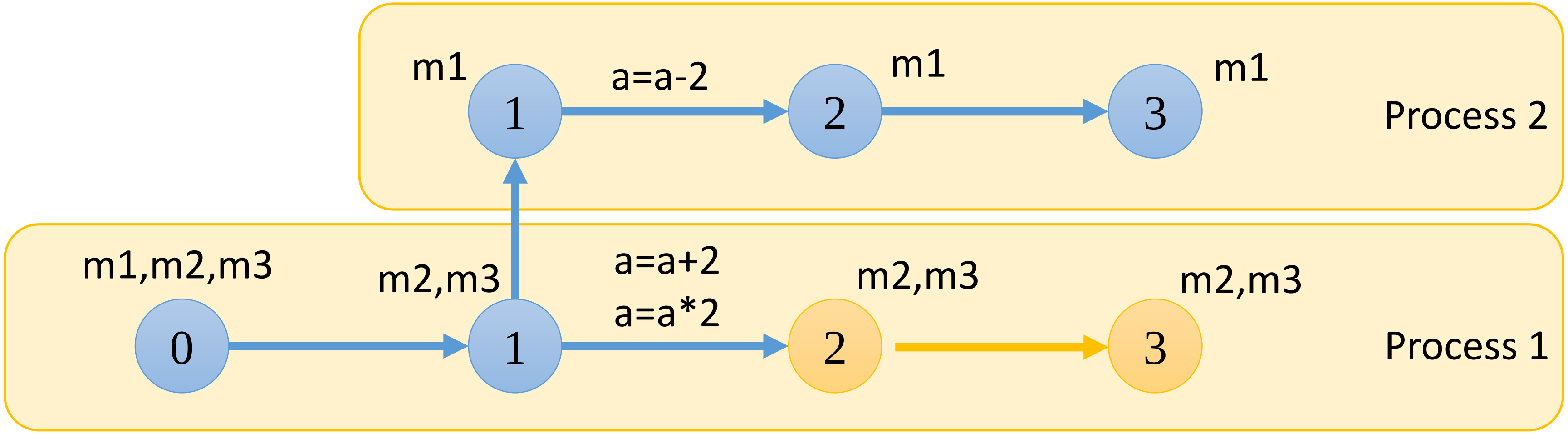
```
1: a=x();
2: a=a*2;
3: y(a);
```



Our Contribution

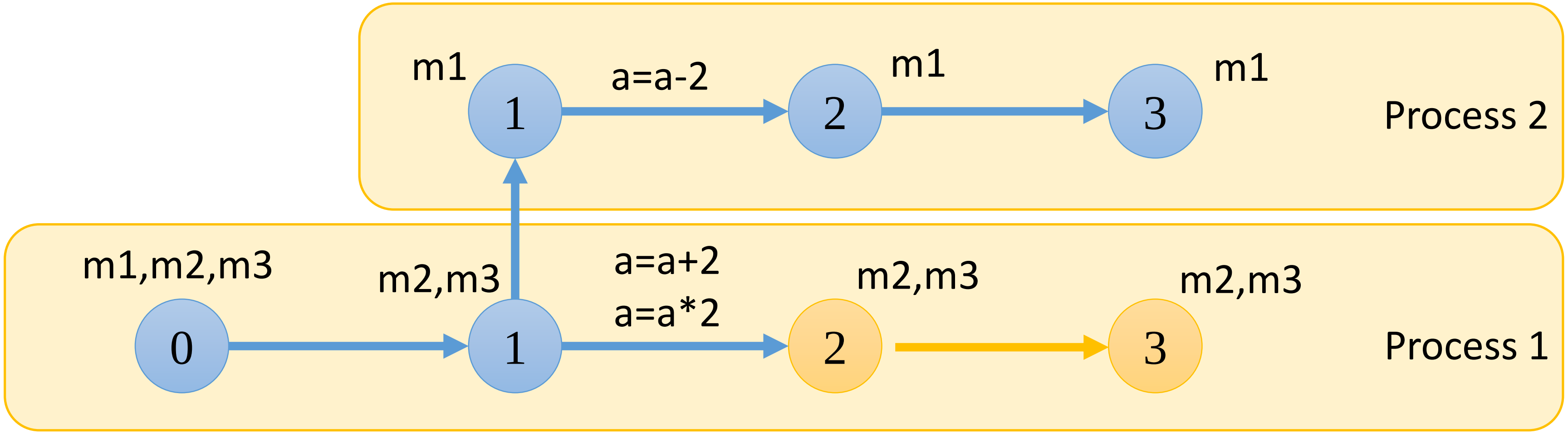
- Equivalence Modulo States
  - Two statements are *equivalent modulo the current state* if executing them leads to the same state from the current state
  - Statements
    - a = a * 2
    - a = a + 2
  - are equivalent modulo
    - State 2 where a == 2

Mutation Analysis via Equivalence Modulo States



- Start with a process representing all mutants
- At each state, group next statements into equivalence classes modulo the current state
- Fork processes and execute each group in one process

Challenges



- Objective: Overheads << Benefits
  - Challenge 1: How to efficiently determine equivalences between statements?
  - Challenge 2: How to efficiently fork executions?
  - Challenge 3: How to efficiently classify the mutants?

Challenge 1: Determine Statement Equivalence

- Performance trial executions of statements and record their changes to states
  - State: a==2
  - a=a+2 $\Longrightarrow \{a \rightarrow 4\}$
  - a=a*2 $\Longrightarrow \{a \rightarrow 4\}$
- Compare their changes to determine equivalence

- Does not work on statements making many changes
  - f(x, y), f(y, x)

Challenge 1: Determine Statement Equivalence

- Record abstract changes that can be efficiently compared
- Ensuring $c(s_1) \neq c(s_2) \implies a(s_1) \neq a(s_2)$
  - s_1, s_2: Statements
  - $c(s)$: Concrete changes made by s
  - $a(s)$: Abstract changes made by s

- Abstract changes of method call: values of arguments
  - State: x = 2, y =2
  - f(x, y) $\Longrightarrow$ <2,2>
  - f(y, x) $\Longrightarrow$ <2,2>

Challenge 2: Fork Execution

- Memory: the POSIX system call “fork()”
  - Implements the copy-on-write mechanism
  - Integrated with POSIX virtual memory management

- Other resources: files, network accesses, databases
  - Solution 1: implement the copy-on-write mechanism
  - Solution 2: map them into memory

Experiments – Mutation Operators

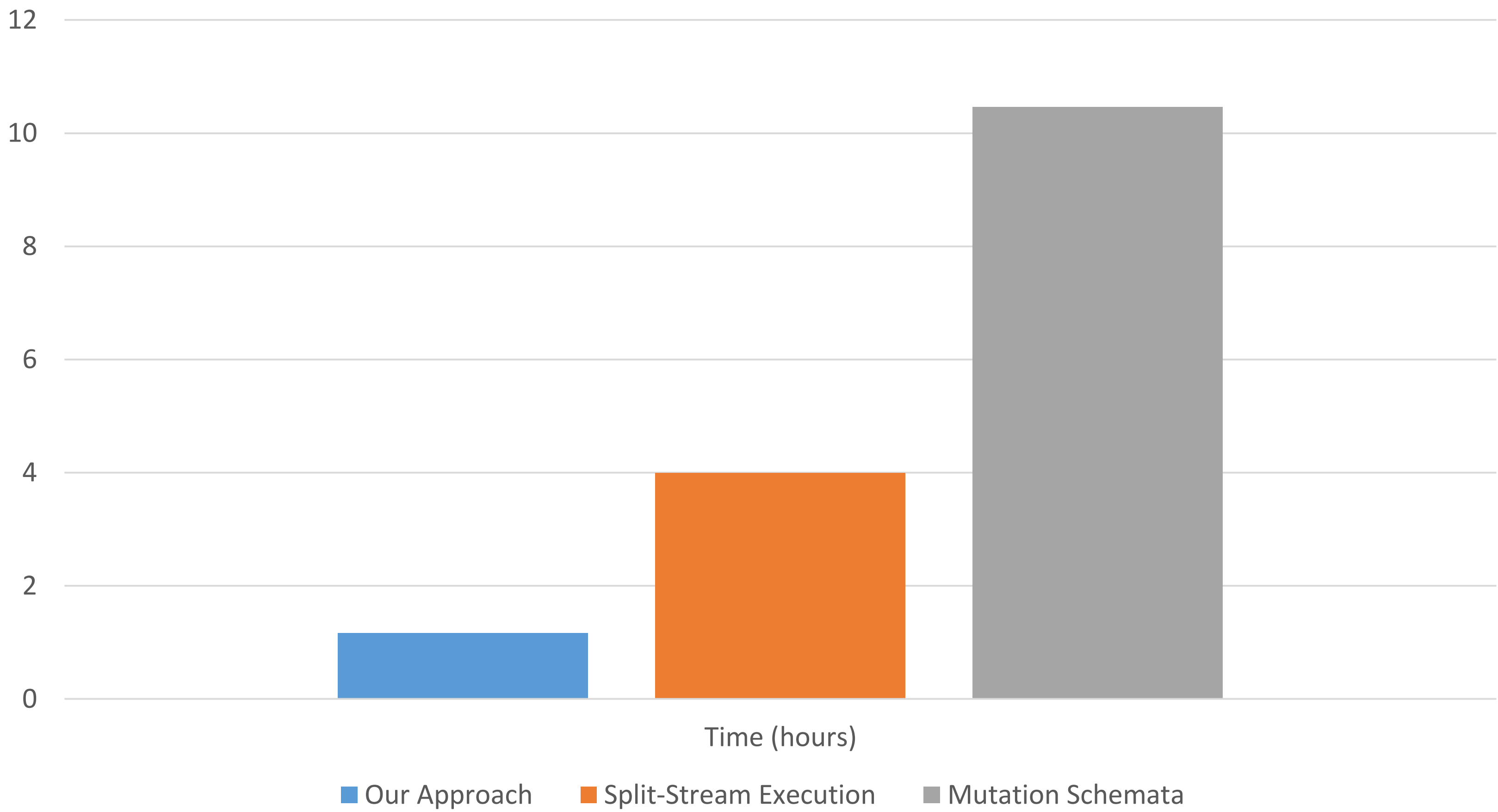
Name	Description	Example
AOR	Replace arithmetic operator	$a + b \rightarrow a - b$
LOR	Replace logic operator	$a \,\&\, b \rightarrow a \mid b$
ROR	Replace relational operator	$a == b \rightarrow a >= b$
LVR	Replace literal value	$T \rightarrow T + 1$
COR	Replace bit operator	$a \,\&\&\, b \rightarrow a \,\|\|\, b$
SOR	Replace shift operator	$a >> b \rightarrow a << b$
STDC	Delete a call	$foo() \rightarrow nop$
STDS	Delete a store	$a = 5 \rightarrow nop$
UOI	Insert a unary operation	$b = a \rightarrow a++;\ b = a$
ROV	Replace the operation value	$foo(a, b) \rightarrow foo(b, a)$
ABV	Take absolute value	$foo(a, b) \rightarrow foo(abs(a), b)$

- Defined on LLVM IR
- Mimicking Javalanche and Major

Experiments - Dataset

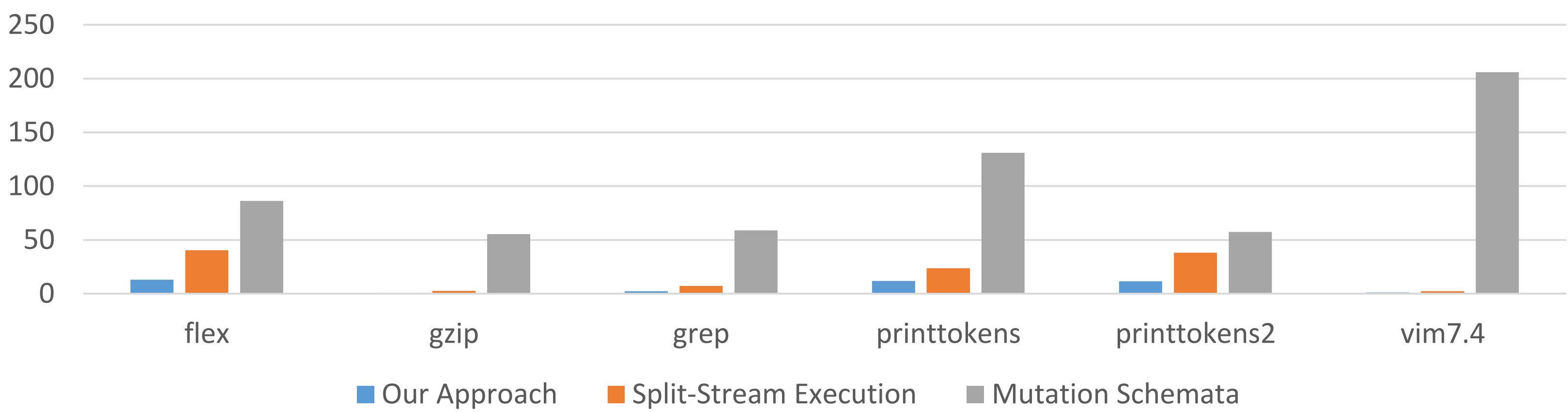
Name	LOC	Tests	Mutants	Locations
flex	10334	42	56916	5119
gzip	4331	214	37326	3058
grep	10102	75	58571	4373
printtokens	475	4130	1862	199
printtokens2	401	4115	2501	207
replace	512	5542	3000	220
schedule	292	2650	493	55
schedule2	297	2710	1077	121
tcas	135	1608	937	73
totinfo	346	1052	756	63
vim 7.4	477257 (42073)	98	173683	14124
Total	504482	20736	337122	27612

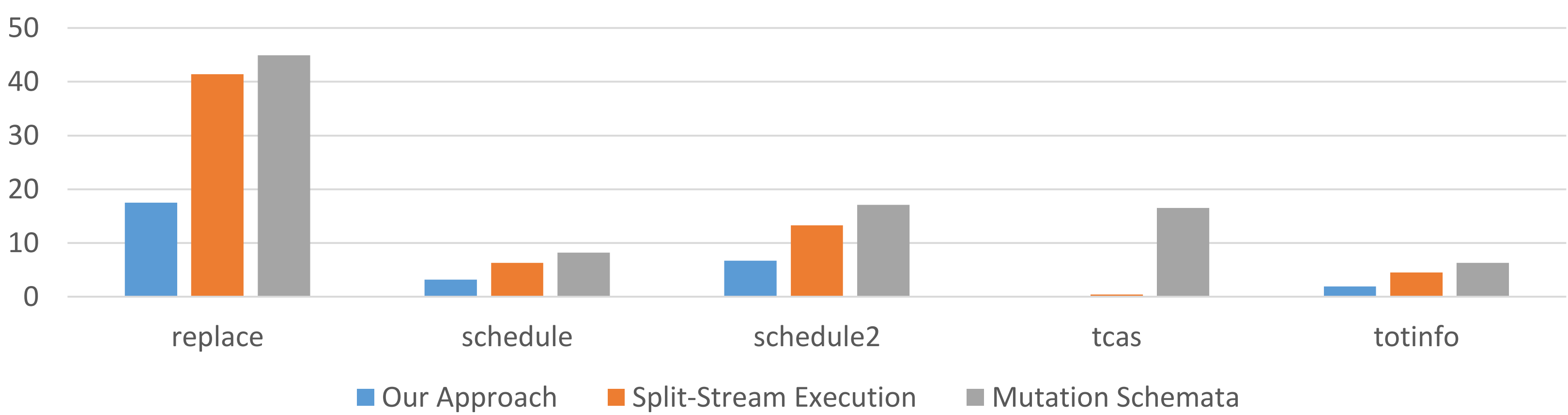
Experiments - Results



2.56X speedup over SSE, and 8.95X speedup over MS

Experiments - Results





Discussion: Why worked?

- Overheads: the overhead for each instruction is *small*
  - Not related to the size of the program, effectively O(1)

- Benefits: equivalences between statements modulo the current state are *common* in mutation analysis
  - $a > b \Rightarrow \begin{array}{c} a \geq b \\ a > b + 1 \\ a > c \\ c > b \end{array}$

- See paper for a detailed study on overheads/benefits

Discussion: Eliminating More Redundancies

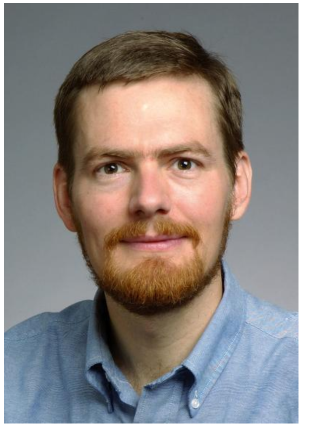
- Translating to model checking problem
  - [Kästner et al., 2012]
  - [Kim, Khurshid, and Batory, 2012]
- Record multiple states as a meta state at variable level
  - [Kästner et al., 2012]
  - [Meinicke, 2014]
- Overheads yet need to be controlled

Conclusion

- Mutation analysis is useful
- Scalability is the a key challenge
- Eliminating redundancy is a promising way to address scalability
- Overhead and benefit must be balanced
- Equivalence modulo states could achieve 2.56X speedup over SSE

Acknowledgments

- We acknowledge **Rene Just** and **Micheal Ernst** for fruitful discussion helping scope the paper



- and ISSTA Program Committee for the recognition
- and you for listening!