OptDebug: Fault-Inducing Operation Isolation for Dataflow Applications

ACM Symposium of Cloud Computing 2021

Muhammad Ali Gulzar and Miryung Kim

Prevalence of Big Data Analytics

Use of large-scale data

Insurance

Advertisement

Finance

...

Data Processing Systems

Apache Spark

Hadoop

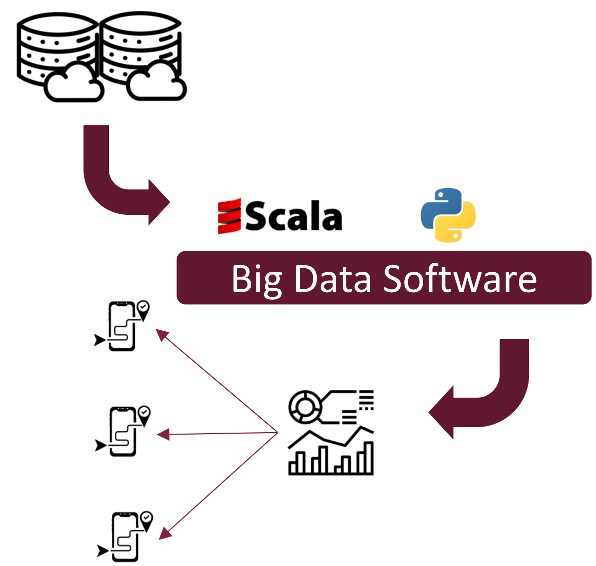
TensorFlow

Hive

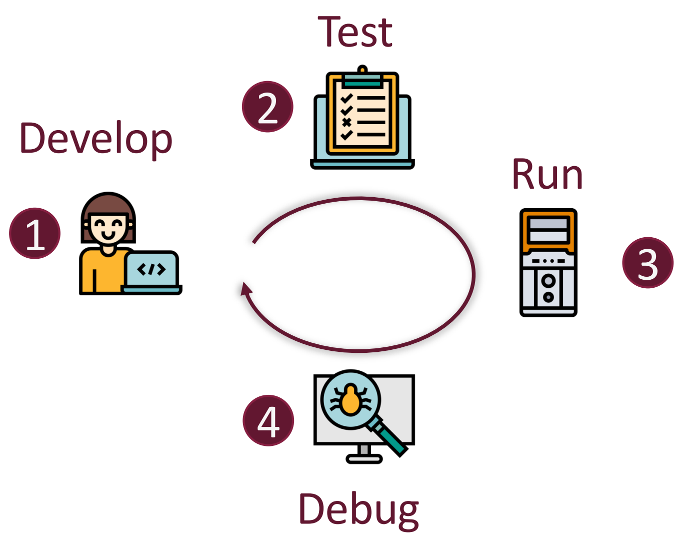
Microsoft Azure

aws

Big Data Applications

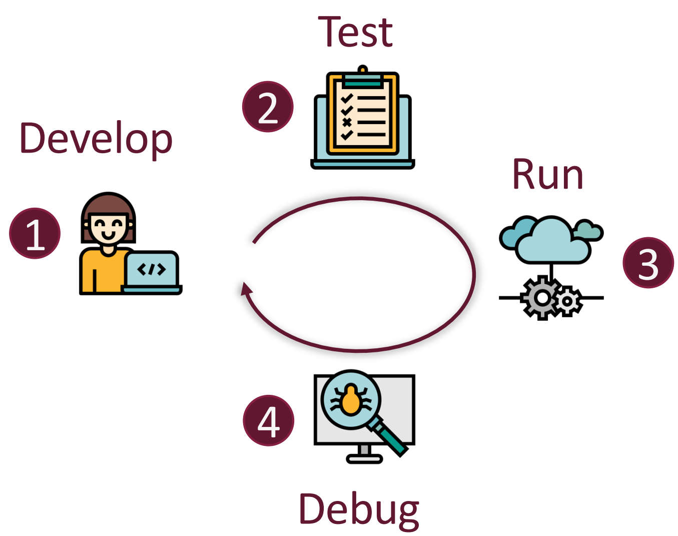


Debugging in Traditional Software



- Debugging is interactive and quick.
- Trial and error is feasible. Each execution takes a few milliseconds.
- Direct access to program states and variables.

Debugging in Dataflow Applications



- Debugging is slow and expensive, mostly via post mortem logs.
- Trial and error is time-consuming and expensive. Each execution takes a a few hours and expensive compute cycles.
- Due to remote, distributed processing, there is no easy, direct access to program states and variables.

Running Example

Calculate the total flying hours for less-than-four hour flights grouped by each departure hour.

```
val log = "s3://IATA-data/logs-2020/transit.log"
val input = new SparkContext(sc).textFile(log)
input.map { s =>
 val tokens = s.split(",")
 val dept_hr = tokens(2).split(":")(0)
 val diff = getDiff(tokens(4), tokens(2))
 (dept_hr, diff) }
 .filter(v => v._2 < 4)
 .reduceByKey(_+_)

// Calculates the difference between time
def getDiff(arr: String, dep: String): Float = {
  val arr_hr = parseHour(arr)
  val dep_hr = parseHour(dep)
  if( arr_hr - dep_hr < 0){ // across midnight
    return arr_hr - dep_hr - 24 }
return arr_hr - dep_hr }
```

Input Dataset

Pass ID	Dep Airport	Dep Time	Arr Airport	Arr Time
XAY993311	CLT	13:15	ORD	15:15
EWS121311	LAX	10:45	ORD	15:00
AAQ591783	SJC	03:33	MNN	8:20

Running Example

```
val log = "s3://IATA-data/logs-2020/transit.log"
val input = new SparkContext(sc).textFile(log)
input.map { s =>
 val tokens = s.split(",")
 val dept_hr = tokens(2).split(":")(0)
 val diff = getDiff(tokens(4), tokens(2))
 (dept_hr, diff) }
 .filter(v => v._2 < 4)
 .reduceByKey(_+_)

// Calculates the difference between time
def getDiff(arr: String, dep: String): Float = {
  val arr_hr = parseHour(arr)
  val dep_hr = parseHour(dep)
  if( arr_hr - dep_hr < 0){ // across midnight
    return arr_hr - dep_hr - 24 }
return arr_hr - dep_hr }
```

Input Dataset

Pass ID	Dep Airport	Dep Time	Arr Airport	Arr Time
XAY993311	CLT	13:15	ORD	15:15
EWS121311	LAX	10:45	ORD	15:00
AAQ591783	SJC	03:33	MNN	8:20

Example: Code Space Debugging is Difficult.

Map

Dep Hour	Flying Hours
13	2.0
10	5.75
03	5.33

```
val log = "s3://IATA-data/logs-2020/transit.log"
val input = new SparkContext(sc).textFile(log)
input.map { s =>
 val tokens = s.split(",")
 val dept_hr = tokens(2).split(":")(0)
 val diff = getDiff(tokens(4), tokens(2))
 (dept_hr, diff) }
 .filter(v => v._2 < 4)
 .reduceByKey(_+_)

// Calculates the difference between time
def getDiff(arr: String, dep: String): Float = {
  val arr_hr = parseHour(arr)
  val dep_hr = parseHour(dep)
  if( arr_hr - dep_hr < 0){ // across midnight
    return arr_hr - dep_hr - 24 }
return arr_hr - dep_hr }
```

Example: Code Space Debugging is Difficult.

Filter

Dep Hour	Flying Hours
~~13~~	~~2.0~~
10	5.75
03	5.33

```
val log = "s3://IATA-data/logs-2020/transit.log"
val input = new SparkContext(sc).textFile(log)
input.map { s =>
 val tokens = s.split(",")
 val dept_hr = tokens(2).split(":")(0)
 val diff = getDiff(tokens(4), tokens(2))
 (dept_hr, diff) }
 .filter(v => v._2 < 4)
 .reduceByKey(_+_)

// Calculates the difference between time
def getDiff(arr: String, dep: String): Float = {
  val arr_hr = parseHour(arr)
  val dep_hr = parseHour(dep)
  if( arr_hr - dep_hr < 0){ // across midnight
    return arr_hr - dep_hr - 24 }
return arr_hr - dep_hr }
```

Example: Code Space Debugging is Difficult.

Reduce

Dep Hour	Flying Hours
11	175080
20	173460
23	-222780

Why is Total Flying Hours negative?

- Data is clean and passes all sanity checks
- Provenance based debugging approaches only debug data-space and not code-space.

```
val log = "s3://IATA-data/logs-2020/transit.log"
val input = new SparkContext(sc).textFile(log)
input.map { s =>
 val tokens = s.split(",")
 val dept_hr = tokens(2).split(":")(0)
 val diff = getDiff(tokens(4), tokens(2))
 (dept_hr, diff) }
 .filter(v => v._2 < 4)
 .reduceByKey(_+_)

// Calculates the difference between time
def getDiff(arr: String, dep: String): Float = {
  val arr_hr = parseHour(arr)
  val dep_hr = parseHour(dep)
  if( arr_hr - dep_hr < 0){ // across midnight
    return arr_hr - dep_hr - 24 }
return arr_hr - dep_hr }
```

Example: Code Space Debugging is Difficult.

Reduce

Dep Hour	Flying Hours
11	175080
20	173460
23	-222780

Why is Total Flying

How can we precisely detect code (i.e., operations or APIs) responsible for a given suspicious or incorrect result?

```
val log = "s3://IATA-data/logs-2020/transit.log"
val input = new SparkContext(sc).textFile(log)
input.map { s =>
 val tokens = s.split(",")
 val dept_hr = tokens(2).split(":")(0)
 val diff = getDiff(tokens(4), tokens(2))
 (dept_hr, diff) }
 .filter(v => v._2 < 4)
```

```
ime
Float = {
```

```
  if( arr_hr - dep_hr < 0){ // across midnight
    return arr_hr - dep_hr - 24 }
return arr_hr - dep_hr }
```

- Data is clean and passes all sanity checks
- Provenance based debugging approaches only debug data-space and not code-space.

Prior Work:
Data Provenance – Data Space Debugging

Map

Dep Hour	Flying Hours
23	2.0
21	5.75
23	5.33
…	…

2

Filter

Dep Hour	Flying Hours
23	2.0
21	5.75
23	5.33
…	…

1

Reduce

Dep Hour	Flying Hours
11	175080
20	173460
23	-222780

3

Input Dataset

Pass ID	Dep Airport	Dep Time	Arr Airport	Arr Time
XAY993311	CLT	23:15	ORD	15:15
EWS121311	LAX	21:45	ORD	04:00
AAQ591783	SJC	23:33	MNN	5:20

Data provenance results of a suspicious output could be large in the order of millions.

Data Provenance – Data Space Debugging

Map

Dep Hour	Flying Hours
23	2.0
21	5.75
23	5.33
…	…

Filter

Dep Hour	Flying Hours
23	2.0
21	5.75
23	5.33
…	…

Reduce

Dep Hour	Flying Hours
11	175080
20	173460
23	-222780

1

2

3

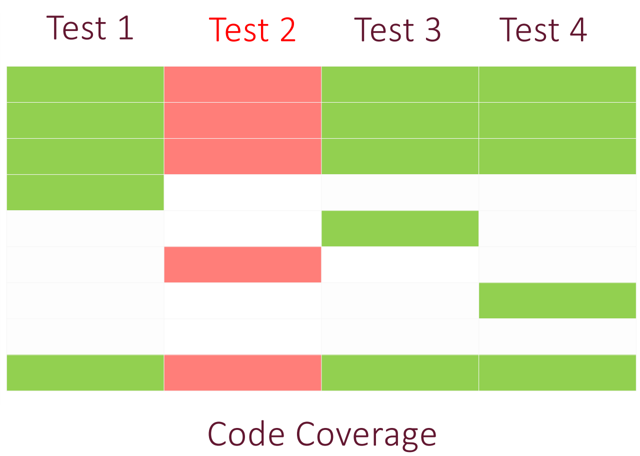
Pass ID				
XAY9933				
EWS121311	LAX	21:45	ORD	04:00
AAQ591783	SJC	23:33	MNN	5:20

Data Provenance techniques identify culprit records in the data-space.
Developers still need to **debug in the code-space**.

Faulty Code Localization

- For traditional software, **spectra-based fault localization** [Jones and Harrold 2002] uses existing test suites to isolate code statements responsible for a test failure.

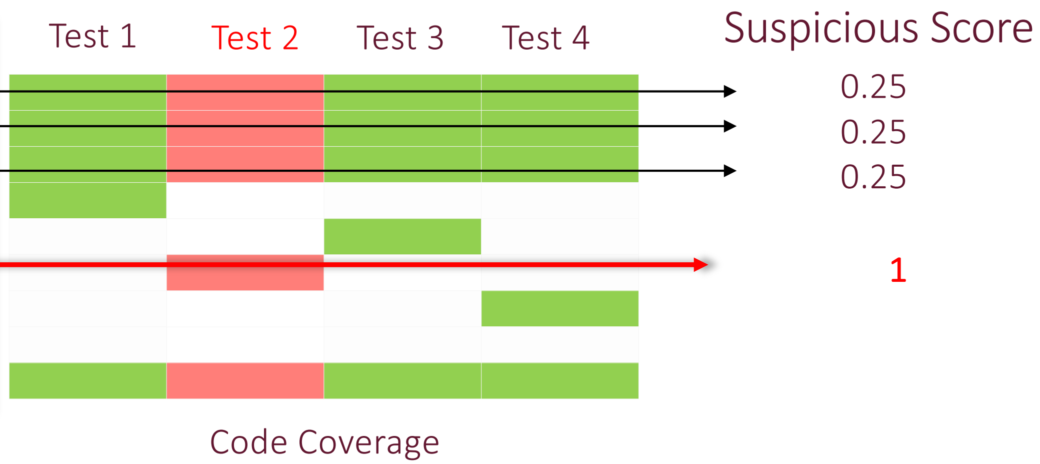
```
//Function to convert temperature measurements
def convert( opt:String, value: Float ) : Float =
{
opt match {
         case “KToC” => value - 273.15
         case “CToK” => value + 273.15
         case “FToC” => (value * 5) / 9
         case “CtoF” => (9/5) * value + 32
          _ => value
}
```



Faulty Code Localization

- For traditional software, **spectra-based fault localization** [Jones and Harrold 2002] uses existing test suites to isolate code statements responsible for a test failure.

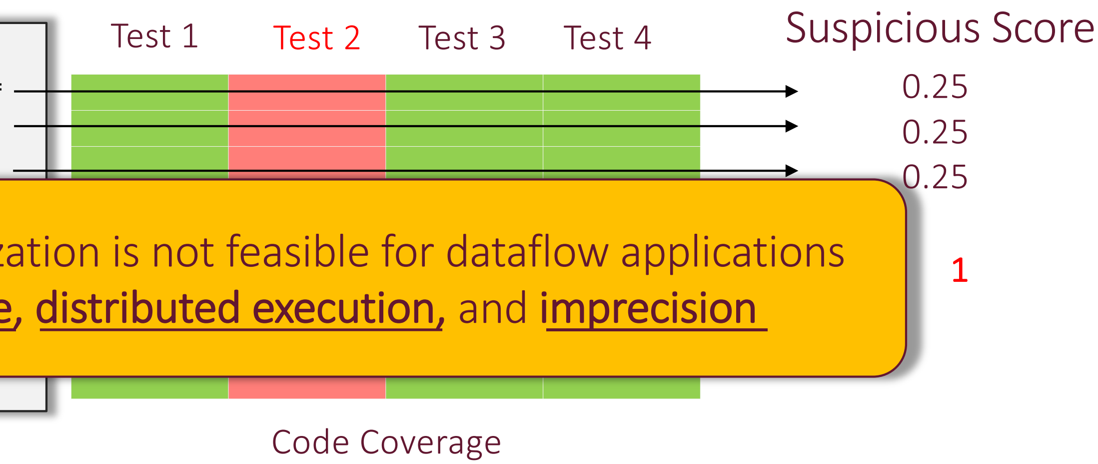
```
//Function to convert temperature measurements
def convert( opt:String, value: Float ) : Float =
{
opt match {
         case “KToC” => value - 273.15
         case “CToK” => value + 273.15
         case “FToC” => (value * 5) / 9
         case “CtoF” => (9/5) * value + 32
          _ => value
}
```



Faulty Code Localization

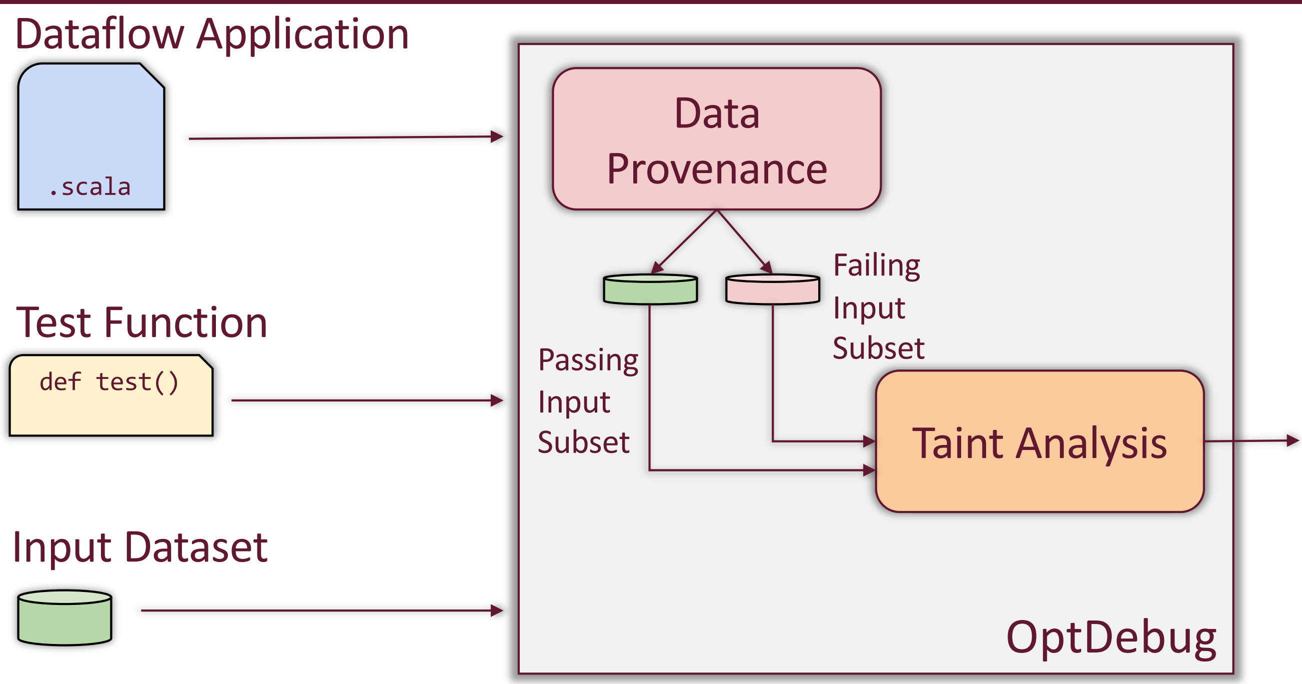
- For traditional software, **spectra-based fault localization** [Jones and Harrold 2002] uses existing test suites to isolate code statements responsible for a test failure.

```
//Function to convert temperature measurements
def convert( opt:String, value: Float ) : Float =
{
opt match {
        case “
        case “
        case “
        case “
         _ =>
}
```



Spectra-Based Fault Localization is not feasible for dataflow applications due to: large input size, distributed execution, and imprecision

OptDebug: Fault Code Localization in DISC



Fault Inducing Operation
Line Number: 33
Operation: Int.minus()

OptDebug precisely pinpoints the operation and code line number that is responsible for a test failure.

Observation 1: Infeasibility of Code Debugging

Input Dataset : 2 billion rows

Pass ID	Dep Airport	Dep Time	Arr Airport	Arr Time
XAY993311	CLT	13:15	ORD	15:15
EWS121311	LAX	10:45	ORD	15:00
AAQ591783	SJC	03:33	MNN	8:20

Code Coverage entries
~ 10X of 2 billion rows

```
val log = "s3://IATA-data/logs-2020/transit.log"
val input = new SparkContext(sc).textFile(log)
input.map { s =>
 val tokens = s.split(",")
 val dept_hr = tokens(2).split(":")(0)
 val diff = getDiff(tokens(4), tokens(2))
 (dept_hr, diff) }
 .filter(v => v._2 < 4)
 .reduceByKey(_+_)
```

Collecting code coverage when running an application on large data is prohibitively expensive.

Insight 1: Test Input Simplification

- Using user-provided test function, we can retrieve simplified passing and failing test input from the dataset.

Output

Dep Hour	Flying Hours
11	175080
21	173460
23	-222780

Data Provenance

Data Provenance

Input Dataset

Pass ID	Dep Airport	Dep Time	Arr Airport	Arr Time
XAY993311	CLT	23:15	ORD	15:15
EWS121311	LAX	21:45	ORD	04:00
AAQ591783	SJC	23:33	MNN	5:20
….	…..	….	…..	….

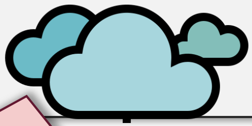
By **reducing data** to only culprit input records, we speed-up spectra-based fault code localization on dataflow applications.

Observation 2: Collection of Code Coverage

Input Dataset : 2 billion rows

Pass ID	Dep Airport	Dep Time	Arr Airport	Arr Time
XAY993311	CLT	13:15	ORD	15:15
EWS121311	LAX	10:45	ORD	15:00
AAQ591783	SJC	03:33	MNN	8:20

```
val log = "s3://IATA-data/logs-2020/transit.log"
val input = new SparkContext(sc).textFile(log)
input.map { s =>
 val tokens = s.split(",")
 val dept_hr = tokens(2).split(":")(0)
 val diff = getDiff(tokens(4), tokens(2))
 (dept_hr, diff) }
 .filter(v => v._2 < 4)
 .reduceByKey(_+_)
```



- Requires JVM instrumentation at each node in the cluster.
- Cannot differentiate between application vs framework code

Traditional coverage tools required system-level modifications to support coverage collection in a distributed setting.

Insight 2: Taint Analysis

- Instead of collecting code coverage at the JVM level, we augment data types with taint containing the history of applied operations.

XAY993311 CLT 13:15 ORD 15:15

```
val log = "s3://IATA-data/logs-2020/transit.log"
val input = new SparkContext(sc).textFile(log)
input.map { s =>
 val tokens = s.split(",")
 val dept_hr = tokens(2).split(":")(0)
 val diff = getDiff(tokens(4), tokens(2))
 (dept_hr, diff) }
 .filter(v => v._2 < 4)
 .reduceByKey(_+_)
```

Variable	Value	Taint (Line Number)
s	XAY993311 CLT 13:15 ORD 15:15	[3]

OptDebug leverages operator overloading and type-inference to capture the code line number at each statement. It is **platform-agnostic**.

Insight 2: Taint Analysis

- Instead of collecting code coverage at the JVM level, we augment data types with taint containing the history of applied operations.

XAY993311 CLT 13:15 ORD 15:15

```
val log = "s3://IATA-data/logs-2020/transit.log"
val input = new SparkContext(sc).textFile(log)
input.map { s =>
 val tokens = s.split(",")
 val dept_hr = tokens(2).split(":")(0)
 val diff = getDiff(tokens(4), tokens(2))
 (dept_hr, diff) }
 .filter(v => v._2 < 4)
 .reduceByKey(_+_)
```

Variable	Value	Taint (Line Number)
dept_hr	13	[3,4,5]

OptDebug leverages operator overloading and type-inference to capture the code line number at each statement. It is **platform-agnostic**.

Insight 2: Taint Analysis

- Instead of collecting code coverage at the JVM level, we augment data types with taint containing the history of applied operations.

XAY993311 CLT 13:15 ORD 15:15

```
val log = "s3://IATA-data/logs-2020/transit.log"
val input = new SparkContext(sc).textFile(log)
input.map { s =>
 val tokens = s.split(",")
 val dept_hr = tokens(2).split(":")(0)
 val diff = getDiff(tokens(4), tokens(2))
 (dept_hr, diff) }
 .filter(v => v._2 < 4)
 .reduceByKey(_+_)
```

Variable	Value	Taint (Line Number)
dept_hr	13	[3,4,5]
diff	2	[3,4,5,7,12,13,14,17]

OptDebug leverages **operator overloading and type-inference** to capture the code line number at each statement. It is **platform-agnostic**.

Observation 3: Statement Coverage’s Imprecision

```
val log = "s3://IATA-data/logs-2020/transit.log"
val input = new SparkContext(sc).textFile(log)
input.map { s =>
 val tokens = s.split(",")
 val dept_hr = tokens(2).split(":")(0)
 val diff = getDiff(tokens(4), tokens(2))
 (dept_hr, diff) }.filter(v => v._2 < 4).reduceByKey(_+_)

// Calculates the difference between time
def getDiff(arr: String, dep: String): Float = {
  val arr_hr = parseHour(arr)
  val dep_hr = parseHour(dep)
// across midnight
  if( arr_hr - dep_hr < 0) return arr_hr - dep_hr - 24
return arr_hr - dep_hr }
```

Numerous Operations In-lined

Traditional statement coverage only captures line coverage thus incapable of identifying faulty operation.

Insight 3: Operation-level Taint Analysis

- OptDebug extend traditional taint analysis to maintain the history of individual operation applied on the data.

```
XAY993311  CLT 13:15 ORD 15:15
```

```
val log = "s3://IATA-data/logs-2020/transit.log"
val input = new SparkContext(sc).textFile(log)
input.map { s =>
 val tokens = s.split(",")
 val dept_hr = tokens(2).split(":")(0)
 val diff = getDiff(tokens(4), tokens(2))
 (dept_hr, diff) }
 .filter(v => v._2 < 4)
 .reduceByKey(_+_)
```

Variable	Value	Taint (Line Number -> *Operation*)
dept_hr	13	[3, 4 -> *split*, 5 -> *split*, 5 -> *idx*]
diff	2	[… 16 -> *Float.gte*, 17 -> *Float.minus*]

By keeping the history of applied code operations, as opposed to the origin of affected data, OptDebug can precisely identify the faulty operation.

Suspicious Score

- Using Tarantula score (default), OptDebug identifies the operation most likely responsible for a test failure.

Output	Taint (Line Number -> *Operation*)
23 -222780	[3, 4 -> *split*, 5 -> *split*, 5 -> *idx*]
13 173460	[… 16 -> *Float.Lt*, 17 -> *Float.minus*]

LOC/Operation	Pass Test	Fail Test	Score
4 -> *split*, 5 -> *split*, 5 -> *idx* . . . 16 -> *Float.Lt*, 17 -> *Float.minus* . .			0.5 0.5 0.5 .. 0.5 1 0.5 ..

Suspicious Score

- Using Tarantula score (default), OptDebug identifies the operation most likely responsible for a test failure.

Output	Taint (Line Number -> *Operation*)
23 -222780	[3, 4 -> *split*, 5 -> *split*, 5 -> *idx*]
13 173460	[... 16 -> *Float.gte*, 17 -> *Float.minus*]

LOC/Operation	Pass Test	Fail Test	Score
4 -> *split*, 5 -> *split*, 5 -> *idx* . . . 16 -> *Float.Lt*, 17 -> *Float.minus* . .			0.5 0.5 0.5 .. 0.5 1 0.5 ..

```
16. // across midnight
17. if( arr_hr - dep_hr < 0) return arr_hr - dep_hr - 24
18. return arr_hr - dep_hr }
```

How well does OptDebug work in Practice?

- We evaluate OptDebug on **6 real-world benchmark** programs
- Input Dataset size ranging from **2 GB to 93 GB**
- Injected fault inspired by prior study on dataflow application faults reported on Stack overflow and Apache Spark mailing lists
- Comparison against baselines
  - Data Provenance
  - Traditional Spectra-based fault localization

RQ1: Fault Localizability

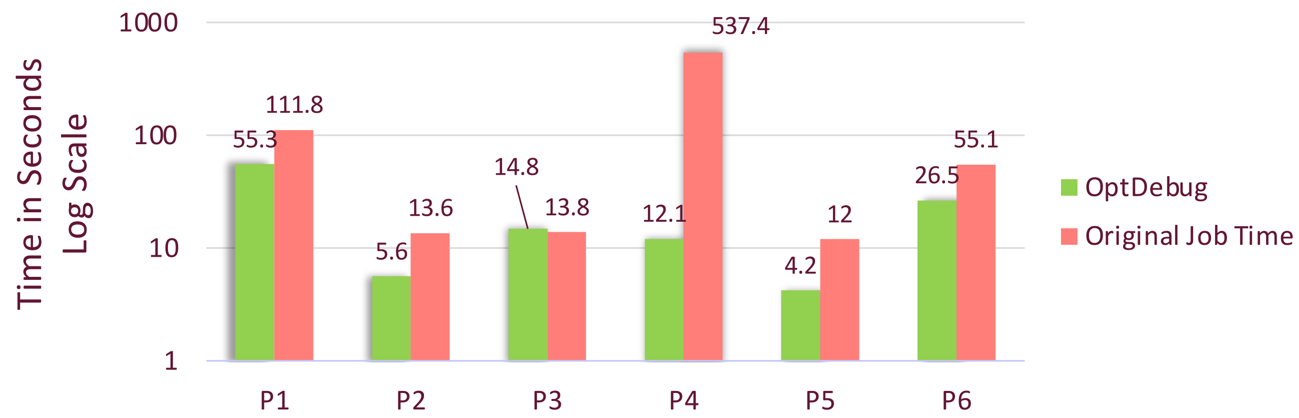
- To evaluate OptDebug's capability to detect code faults, we measure how precisely and accurately OptDebug finds faulty code lines (/operations) in the subject programs.

Program	Input Row Count	Simplified Input via DP	Known Faults	Detected Faults
P1	10^8	1.7×10^6	2	2
P2	10^7	4.5×10^4	1	2
P3	10^7	2.2×10^5	2	3
P4	10^9	1.9×10^5	1	1
P5	10^7	210	1	1
P6	10^9	7.0×10^5	2	2

OptDebug finds the fault-inducing operation with 86% precision and 100% recall on average.

RQ2: Debugging Time

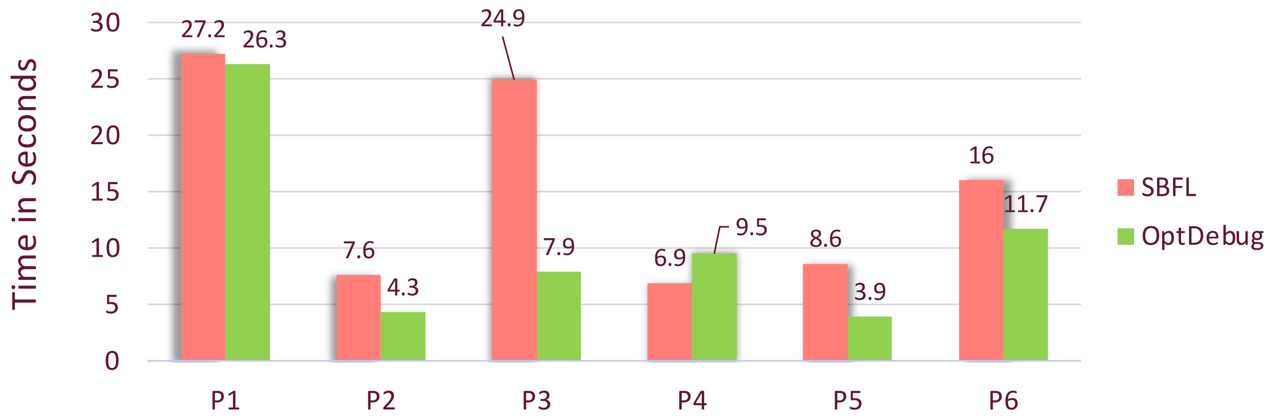
- We measure the time OptDebug takes to find the fault-inducing operation (i.e., time taken after a given application produces a failing outcome defined via a test predicate).



OptDebug finds the fault-inducing operation with 86% precision and 100% recall on average.

RQ3: Taint Analysis vs. SBFL

- We compare OptDebug's operation-level taint analysis on running spectra-based fault localization with a simplified input.



OptDebug's taint analysis on a simplified input is on average 27% faster than applying spectra-based fault localization.

Conclusion

- OptDebug proposes a novel operation-level taint analysis to track the history of executed code lines and APIs to automatically determine the root cause in terms of code lines and API operations.
- OptDebug is a library (jar) that can be imported in any Apache Spark application written in Scala.

https://github.com/maligulzar/OptDebug