Fault Localization Using Value Replacement

Dennis Jeffrey
Neelam Gupta
Rajiv Gupta
ISSTA '08

Presented by Amy Siu 12/9/08

What is fault localization?

- Software building is a human-intensive process
  - ➔ Prone to error

- Software debugging consists of two phases
  - Locating the error
  - Fixing the error

Why is fault localization difficult?

- The point of failure may appear anywhere after the faulty code statement
- The faulty code statement is not always obvious

- **Manual** inspection requires
  - Human effort
  - Code familiarity
  - Domain knowledge
- **Automatic** localization is still an open research problem
  - Computationally intensive

Problem statement

- Investigate an alternative method to localize faulty code statements
  - Automated
  - Computationally less intensive
  - Able to locate faulty code statement even if the point of failure occurs after that statement

State of the art - dynamic analysis

- **Dynamic program slicing**
  - Computationally expensive and not scalable
- **Delta debugging**
  - Cause effect chains have less granularity
- **Nearest neighbor**
  - Poorer performance compared to the rest of the state-of-the-art
- **Statistical technique**
  - Most similar approach to proposed technique

State of the art - *Tarantula*

- *Tarantula* by Jones and Harrold '05 is the **closest statistical technique** in the state-of-the-art to the proposed technique
- **Baseline** for this paper
- Evaluated over the **same** Siemens benchmark suite

How does *Tarantula* work?

- **Intuition:** statements executed primarily by failing runs are more likely to be faulty
- **Keep track** of statements in successful and failing runs
- **Rank** statements based on statistics

Definitions

- **Value mapping**
  - Variables: concrete value, e.g. x = 10
  - Predicate statements: branch outcome, e.g. “else” branch
- **IVMP** (Interesting Value Mapping Pair)
  - A pair of value mappings
  - **Original** value mapping exists in failing run with **wrong output**
  - **Alternate** value mapping causes the output to become **correct**
- **Value profile**
  - All value mappings for a program
  - Each mapping may contribute to original, alternate, or both types of IVMPs

IVMP algorithm

Step 1: initialize value profile

Step 2: search for IVMPs

Running time = O(t*m)

- t = # statements
- m = max. # alt. mappings

```
input:
    Faulty program P, and failing test case f (with
    actual and expected output) from test suite T.
output:
    Set of identified IVMPs for f.
algorithm SearchForIVMPs
Step 1: [Compute value profile for P w/ respect to T
    valProf := {};
    for each test case t in T do
        trace := trace of value mappings from
            execution of t;
        augment valProf using the data in trace;
    end for
Step 2: [Search for IVMPs in f]
    trace_f := trace of value mappings from
        execution of f;
    for each statement instance i in trace_f do
        origMap := value mapping from trace_f at i;
        s := the statement associated with instance i
        for each altMap in valProf at s do
            execute f while replacing origMap
                with altMap at i;
            if output of f becomes correct then
                output IVMP (origMap, altMap) at i
        end for
    end for
end SearchForIVMPs
```

Figure 1: General algorithm for computing IVMPs with respect to a failing run and its test suite.

IVMP Example 1

IVMP at a faulty statement

```
argc := ...;
if (argc < 3) /* 3 should actually be 4 */
    print ("Too few");
else
    print ("Okay");
```

Test Case	Input Values	Actual Output	Expected Output	Result
A	$argc = 2$	Too few	Too few	PASS
B	$argc = 3$	Okay	Too few	FAIL
C	$argc = 4$	Okay	Okay	PASS

Figure 2: Code fragment and test suite based on schedule faulty version v9.

IVMP Example 2

IVMP directly linked to a faulty statement

```
    AltLayVal := ...;
    Pos_RA_Alt_Thresh[0] = 400;
    Pos_RA_Alt_Thresh[1] = 550; /* Should be 500 */
    Pos_RA_Alt_Thresh[2] = 640;
    Pos_RA_Alt_Thresh[3] = 740;
    ...
    if (Pos_RA_Alt_Thresh[AltLayVal] < 525)
        print (0);
    else
        print (1);
```

Test Case	Input Values	Actual Output	Expected Output	Result
A	$AltLayVal = 0$	0	0	PASS
B	$AltLayVal = 1$	1	0	FAIL
C	$AltLayVal = 2$	1	1	PASS

Figure 3: Code fragment and test suite based on tcas faulty version v7.

IVMP Example 3

IVMP in the presence of erroneously omitted statements

```
   int foo(int x, int y)
1:     /* if (y < 0) return x; */
2:     if (y == 0) return 0;
3:     return x + 1;
```

Test Case	Input Values	Actual Output	Expected Output	Result
A	$(x,y) = (1,-1)$	2	1	FAIL
B	$(x,y) = (2,2)$	3	3	PASS
C	$(x,y) = (0,1)$	1	1	PASS

Figure 4: Code fragment and test suite inspired by schedule2 faulty version v1.

Dependence cause vs. Compensation cause

- **Dependence cause**
  - Different statements in the same **definition-use chain**
  - Applying IVMP to **any** statement corrects the error
  - But only **one** statement is the root cause
- **Compensation cause**
  - **Unrelated** statements
  - Applying IVMP to **any** statement also corrects the error
  - The paper does not further discuss details

Ranking statements using IVMPs

```
1:  read (x,y);
2:  a := x + y; /* should be x - y */
3:  if (a < y)
4:      z := a;
5:  else
6:      z := a + 1;
7:  print (z);
```

Test Case	Input Values	Actual Output	Expected Output	Result
A	(x,y) = (0,0)	1	1	PASS
B	(x,y) = (-1,0)	-1	-1	PASS
C	(x,y) = (1,1)	3	1	FAIL
D	(x,y) = (0,1)	1	-1	FAIL

Figure 5: Example with test suite to motivate the need for considering multiple failing runs when ranking statements using IVMPs.

Line 2 is more likely to be faulty than lines 4 and 6

Suspiciousness

Proposed metric: **suspiciousness**

$$suspiciousness(s) := |\{f : f \in F \wedge s \in STMT_{IVMP}(f)\}|$$

Tie-breaker metric: suspiciousness as per ***Tarantula***

$$suspiciousness_{tarantula}(s) = \frac{\frac{failed(s)}{totalFailed}}{\frac{passed(s)}{totalPassed} + \frac{failed(s)}{totalFailed}}$$

Ordering failing runs algorithm

Step 1: find IVMPs

Step 2: use IVMPs to rank statements

 # Re-executions = O(f*t*m)

- f = # failing runs
- t = # statements
- m = max. # alt. mappings

But can **limit** statement instances and alternative mappings – use **shortest** failing runs first

```
input:
    Faulty program P, and test suite T containing
    a set F of failing runs.
output:
    A ranked list of statements exercised by tests in F.
algorithm IVMPBasedStatementRank
begin
Step 1: [Compute IVMPs for each test in F]
    valProf := construct value profile for P wrt. T;
    sort the tests in F in increasing order of trace size;
    workingList := the set of stmts exercised by the
        first failing test case in sorted F;
    for each test f in F taken in sorted order do
        trace_f := stmt instances executed by f;
        for each stmt instance i in trace_f do
            s := the stmt associated with instance i;
            if s not in workingList then continue;
            altMap := alt. mappings for s in valProf;
            altMap_red := subset of altMap with min/max
                values < and > the orig values used at i;
            for each alt. mapping m in altMap_red do
                if s has an IVMP in f then break;
                if applying m at i corrects f's output then
                    report a found IVMP at s in f;
            endfor (each alt mapping)
        endfor (each stmt instance)
        if f has at least one IVMP then
            remove stmts from workingList that are not
                associated with any IVMP in f;
    endfor (each failing run)
Step 2: [Use IVMPs to rank program statements]
    stmts := set of stmts exercised by tests in F;
    for each stmt s in stmts do
        compute suspiciousness(s);
        compute suspiciousness_tarantula(s);
    endfor
    stmts_ranked := sort stmts by suspiciousness,
        break ties by suspiciousness_tarantula;
    output stmts_ranked;
end IVMPBasedStatementRank
```

Figure 6: Our IVMP based statement ranking approach using a reduced IVMP search.

Summary of proposed technique

1. Gather successful and failing **runs**
2. Establish **value profile**
3. Search for **IVMPs**
4. **Rank statements** using suspiciousness
5. **Break ties** with *Tarantula*'s suspiciousness

Experiment 1 – implementation

Valgrind infrastructure

- Synthetic, simulated CPU
- Machine-level instructions
- Value mappings manipulated at the machine instruction level

Machine profile

- Dell PowerEdge 1900 server
- 2 Intel Xeon quad-core processors at 3.00GHz
- 16 GB RAM
- No parallel processing

Experiment 1 – subject programs

***Siemens* benchmark** suite

- All faults are seeded
- At least 5 successful and 5 failing runs

Prog. Name	LOC	# Ver.	Avg. Suite (Pool) Sizes	Program Description
tcas	138	41	17 (1608)	altitude separation
totinfo	346	23	15 (1052)	statistic computation
sched	299	9	20 (2650)	priority scheduler
sched2	297	9	17 (2710)	priority scheduler
ptok	402	7	17 (4130)	lexical analyzer
ptok2	483	9	23 (4115)	lexical analyzer
replace	516	31	29 (5542)	pattern substituter

Table 1: The Siemens benchmark programs. From left to right: program name, # lines of code, # faulty versions, average suite size (and test case pool size), and description of program functionality.

Experiment 1 – compared approaches

5 approaches compared in the experiment

- **IVMP**
- **Tarantula**
- **Tarantula-Pool** – use entire test case pool to get larger test suite
- **IVMP-1** – use only 1 failing run to search IVMPs with
- **IVMP-2** – use 2 failing runs to search IVMPs with

Assign a **score** to ranked statements

$$score(S) = \frac{totalStmtsEx - r}{totalStmtsEx} \times 100\%$$

- Higher score → more statements executed by failing runs are ignored before faulty statement is found

Results – effectiveness

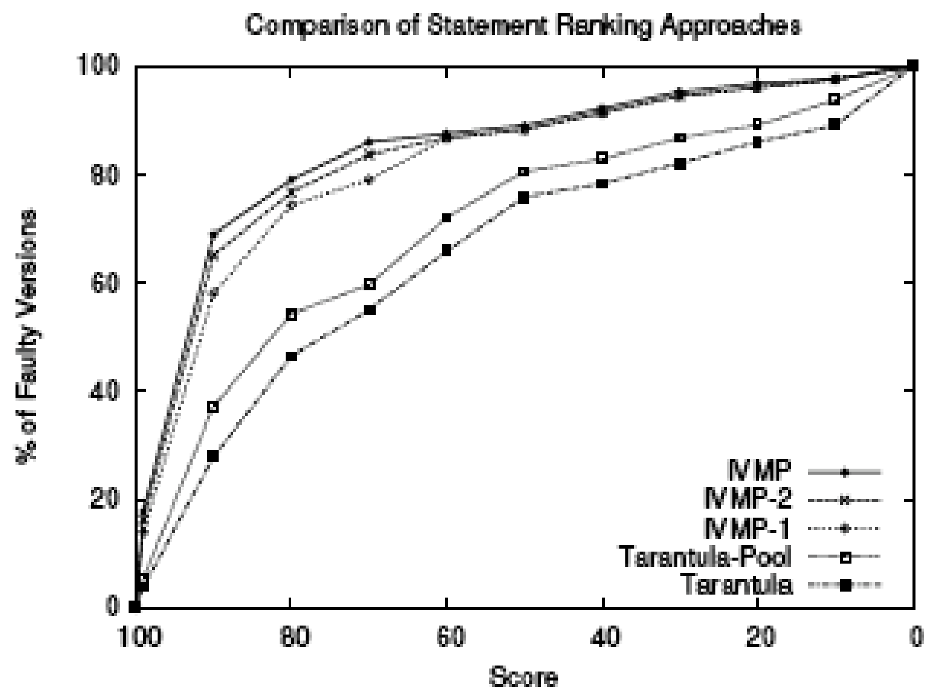

Figure 7: Comparison of statement ranking approaches

- **IVMP** ranks faulty statements higher than *Tarantula*
- **Larger test suite pool** help rank faulty statement higher
- **More failing runs** help rank faulty statements higher

Results – efficiency (I)

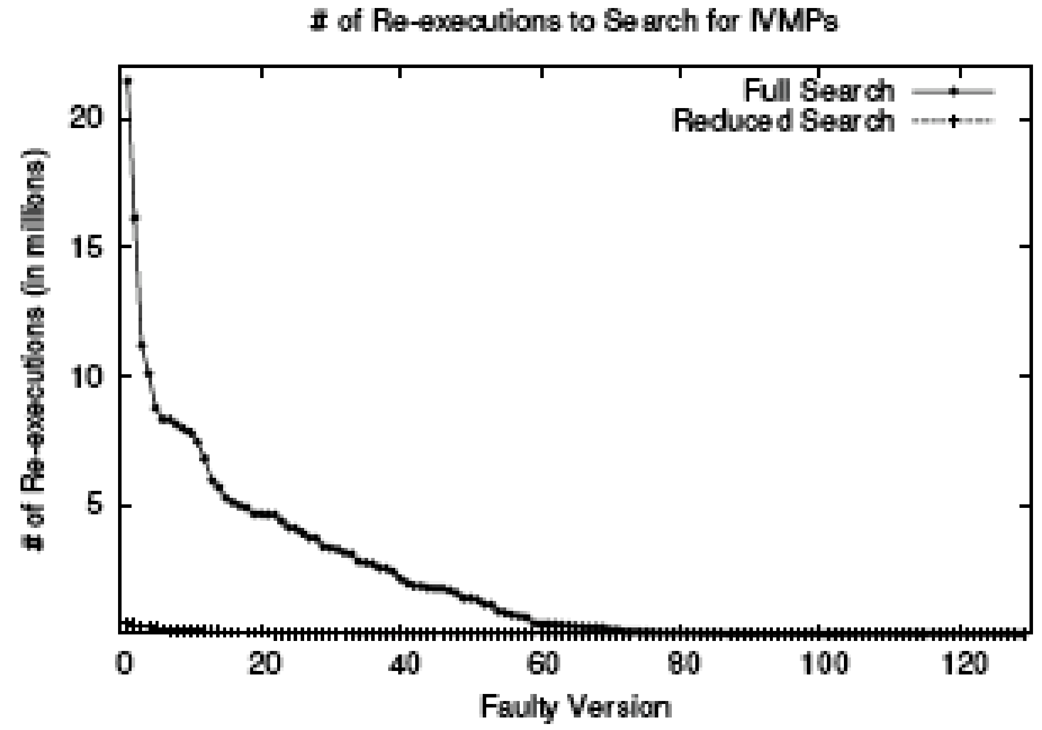

Figure 8: For each faulty version, the number of re-executions (in millions) required for the full and reduced IVMP searches in the IVMP approach.

- Compare variations within IVMP search algorithm
- **Reduced IVMP search** technique drastically reduces # re-executions

Results – efficiency (II)

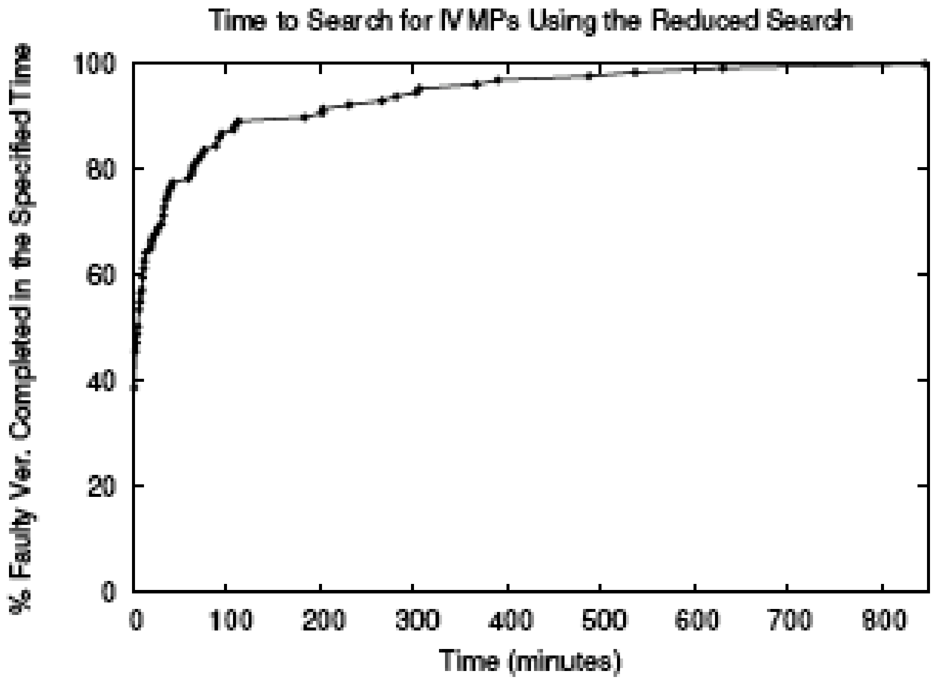

Figure 9: The percentage of faulty versions in which our reduced search for IVMPs is able to complete in the specified amount of time in the IVMP approach.

- Almost 90% of faulty versions have all IVMPs searched **under 100 minutes**
- Maximum time of 840 minutes due to **unusual case** – long failing runs cannot limit IVMP search

Results – efficiency (III)

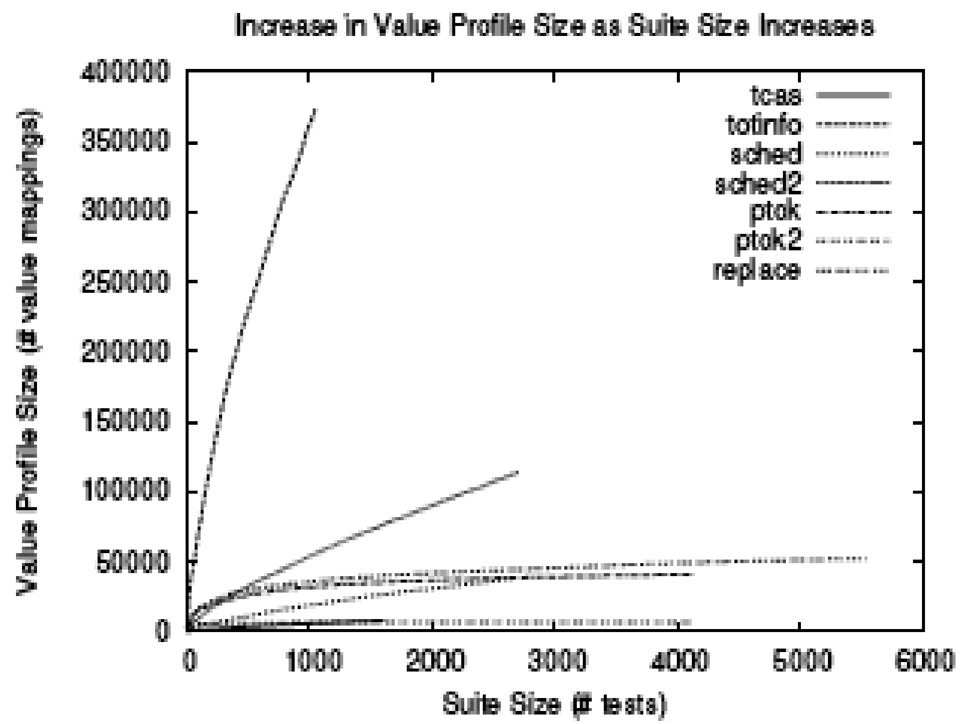

Figure 10: Increase in value profile size as suite sizes increase, for each subject program.

- **Size of value profile** increases logarithmically to test suite size
- **Unusual case** – difficult to pinpoint exact floating point values

Experiment 2 – larger programs

Prog. Name	LOC	Fault Type	Program Description
space	6199	real	ADL interpreter
grep-2.5	5812	real	pattern matcher
sed-4.1.5	12972	seeded	stream editor
flex-2.5.1	10013	seeded	lexical analyzer generator
gzip-1.3	5166	seeded	file compressor

Table 3: Larger subject programs.

Program Name	Faulty Stmt Rank	IVMP Search Time	# Re-executions Done/Possible for IVMPs
space	Tarantula: 106 IVMP: 5	79.5 min	35841/1061154 (3.4%)
grep-2.5	Tarantula: 213 IVMP: 3	0.8 min	241/588 (41.0%)
sed-4.1.5	Tarantula: 35 IVMP: 3	1.8 min	881/5816 (15.1%)
flex-2.5.1	Tarantula: 45 IVMP: 1	0.5 min	87/228 (38.2%)
gzip-1.3	Tarantula: 96 IVMP: 1	215.6 min	126845/6918816 (1.8%)

Table 4: Experimental results using the larger programs (one fault and test suite per program).

Second experiment ran on **5 larger subjects programs**

- **Similar IVMP search time** to experiment 1
- Search time depends on length of **shortest failing trace**, not program size
- Proposed technique is **scalable**

Discussion

Scalability

- Further enhance scalability by limiting IVMP search
- Combine other techniques such as program slicing

Multiple simultaneous faults

- Difficult to find IVMPs that influence each other, or have different effects on program output
- Diminishes effectiveness of proposed approach

Addess values

- Ignored by proposed approach

Conclusions

Proposed IVMP approach is

- **More effective** than the best technique in the state-of-the-art, *Tarantula*
- **Scalable**

Limitations and future work – noted by authors

Limitations

- Only find faults that can be detected via value replacement
- Multiple simultaneous faults
- Address values

Future work

- (No explicit future work section in the paper)
- Combine proposed technique with program slicing to limit IVMP search

Limitations and future work – class discussion

Limitations

- Indirectly linked faulty statements
- Extraneous statements causing a fault – no example, unclear how that works
- “Fuzzy” values generate huge value profile *a la* floating point example
- Dependent on existing runs – both successful and failing ones – to generate rankings

Future work

- Adapt proposed technique in practical environment without machine instruction-level simulator
- Try new technique on even larger programs
- How to use proposed technique when there are no existing test runs to extract value profile from