Efficient Concolic Testing of MPI Applications

Hongbo Li
Zizhong Chen    Rajiv Gupta

CC’19, Washington DC, USA
Feb. 17, 2019
It Is Popular

- Programming languages:
  - Binary machine code, C, Java, and JavaScript.

- Application types:
  - Web applications, sensor network applications, Unix utilities, database applications, and embedded software, GPU programs, image processing software, and so on.

- Various tools:
  - KLEE, DART, SAGE, PEX, jCute, CREST, Jalangi, etc.
COMPI [ IPDPS 2018 ]

- COMPI is a concolic testing tool for MPI programs with following major features:
  - Deals with basic MPI semantics
  - Deals with high testing cost caused by input values, parallelism, and loops

- COMPI achieves 69-86% branch coverage within a few hours
Concolic Testing

(1) Instrumentation

```c
int x;
mark_symbolic(x);
...
__load_symbol(x);
__load_value(0);
__apply('>');
if (x > 0) {
    __log_constraint("x > 0");
    ...
} else {
    __log_constraint("x <= 0");
    ...
}
```

(2) Iterative testing

test inputs: # procs, focus, marked_variables

Instrumented Program

Test Engine

constraints & branch coverage
Issues of COMPI

Our Solutions

Evaluation
Issues of COMPI

› Input generation does not guarantee cost-effective testing

› Floating point data types and operations are not supported
Issue I

- Larger input values $\rightarrow$ Longer execution

- Solution of COMPI: Input Capping

\[ \forall x \leq CAP \]

Program execution $\rightarrow$ Solver

Constraints $\rightarrow$ values
Example

```c
main () {
    MPI_Init();
    // n denotes the width
    // of square matrices
    int n;
    if (n <= 100)
        small_matrix_multi();
    else
        large_matrix_multi();
    MPI_Finalize();
}
```

An MPI program performing matrix multiplication.
main () {
    MPI_Init();
    // n denotes the width
    // of square matrices
    int n;
    if (n <= 100)
        small_matrix_multi();
    else
        large_matrix_multi();
    MPI_Finalize();
}

1st run (n ← 10)

\[ n \leq 100 \]

generate constraints

\[ n > 100 \]

negate a constraint

solve

NO solution

\[ CAP = 50 \rightarrow \text{Fail to cover the else branch} \]
Big Cap

```c
main () {
    MPI_Init();
    // n denotes the width
    // of square matrices
    int n;
    if (n <= 100)
        small_matrix_multi();
    else
        large_matrix_multi();
    MPI_Finalize();
}
```

\[ CAP = 500 \rightarrow \text{High testing cost} \]
No Input Capping

```c
main () {
    MPI_Init();
    // n denotes the width
    // of square matrices
    int n;
    if (n <= 100)
        small_matrix_multi();
    else
        large_matrix_multi();
    MPI_Finalize();
}
```

1st run ($n \leftarrow 10$)  
- generate constraints
  - $n \leq 100$
- negate a constraint
  - $n > 100$
- solve

$n \leftarrow 1234567$

No Capping $\Rightarrow$ Execution failure
Issue II

- Floating-point data types and operations are not supported
Unable to mark $b$

- Fixing it to a value ➔
  either $f1$ or $f2$ could not be explored

Unable to record $a \times 1.1$

- Symbolic representation of $c$ is not existing ➔
  either $f3$ or $f4$ could not be explored
Issues of COMPI

Our Solutions

Evaluation
Our Solutions

- Input tuning → cost effective testing

- Floating-point extension → exploration of branches related to the use of floating-point arithmetic
Our Solutions

› Input tuning ➔ cost effective testing

› Floating-point extension ➔ exploration of branches related to the use of floating-point calculations
  › Constraint solving using reals instead of floating-point numbers ➔ faster constraint solving
Input Tuning

```c
main () {
    MPI_Init();
    // n denotes the width
    // of square matrices
    int n;
    if (n <= 100)
        small_matrix_multi();
    else
        large_matrix_multi();
    MPI_Finalize();
}
```

1st run (n←10)  
- generate constraints
  - n <= 100

2nd run (n←101)  
- use the value

- negate a constraint
  - n > 100

- tune the value
  - (n←1234567)
- solve

UCR
Input Tuning

Binary search of $upper$ in $(0, 1234567)$ satisfying:
\[
\{n > 100\} \cup \{n \leq upper\} \text{ is solvable}
\]
AND \[
\{n > 100\} \cup \{n \leq upper - 1\} \text{ is unsolvable}
\]

$n \leftarrow 1234567$

Solution by solver for $\{n > 100\}$

$n \leftarrow 101$

Tuned solution
Input Tuning for Multi-variable Multi-constraint Case

Need to solve \{constraint1, constraint2\}
Input Tuning for Multi-variable Multi-constraint Case

\[ \text{if } (x - y > 100) \]
\[ \text{if } (y > 0) \]
\[ \text{constraint1} \]
\[ \text{constraint2} \]

Need to solve \( \{x - y > 100, y > 0\} \)
Input Tuning for Multi-variable Multi-constraint Case

Stage I Tuning

\{x = 4321, y = 1234\}

\{x = 4321, y = 1234, x \leq upper1, y \leq upper1\}

\text{min}\{upper\} = 102

Stage II Tuning

\{x = 4321, y = 1234, x \leq upper1, y \leq upper2\}

\text{min}\{upper2\} = 1

Tuning for \{x - y > 100, y > 0\}

\{x = 102, y = 1\}
Input Tuning -- Summary

- Stage I avoids too large values being generated for **ALL** variables appearing in dependent constraints.

- Stage II ensures the smallest value is generated for the **SINGLE** variable appearing in the target constraint based on Stage I.
Our Solutions

› Input tuning ➔ cost effective testing

› Floating-point extension ➔ exploration of branches related to the use of floating-point arithmetic
Floating-Point Extension

- Two floating-point data types supported: *float*, *double*

- The extension adopts the design methodology of symbolic reasoning for integers
  - Instrument floating-point operations
  - Records only *linear constraints*
    - Non-linear constraints are simplified using concrete values, e.g., $x \times y$ is recorded as $C \times x$ with $C$ being the concrete value of $y$
Floating-Point Extension

- Two solvers: Real v.s. Float
  - Accuracy: Real < Float
  - Solving speed: Real > Float

Real v.s. Float based on 100 iterative tests of a synthetic program that compares expression $e$ with constant 0.

<table>
<thead>
<tr>
<th></th>
<th>$e =$</th>
<th>$x$</th>
<th>$x + y$</th>
<th>$x + y + z$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost (float)</td>
<td>31.4s</td>
<td>75.0s</td>
<td>91.2s</td>
<td></td>
</tr>
<tr>
<td>Cost (real)</td>
<td>8.2s</td>
<td>8.1s</td>
<td>8.2s</td>
<td></td>
</tr>
</tbody>
</table>

3.8-11.1X faster
Limitations of COMPI

Our Solutions

Evaluation
Evaluation

- Input tuning is evaluated using HPL, IMB-MPI1, and SUSY-HMC
- Floating-point extension is evaluated using SUSY-HMC
- One hour testing at each configuration
- Initial input values are 1 for all variable in the first test
- In the evaluation of input capping, we selects the same cap for all variables
Input Tuning on HPL

- **Tuning**
  - (539s, 1865)

- **Capping (c = 2)**
  - (3563s, 1866)

- **Capping (c = 8)**
  - (3111s, 1544)

- **None**
  - (3573s, 1840)
Input Tuning on IMB-MPI1

Tuning

- (439s, 766)

Capping (c = 2)

- (964s, 704)

Capping (c = 8)

- (1611s, 733)

None

- (2322s, 735)
Input Tuning on SUSY-HMC

- Tuning
  - # branches vs. time (s)
  - (31492s, 1662)

- Capping (c=2)
  - # branches vs. time (s)
  - (100s, 713)

- Capping (c=8)
  - # branches vs. time (s)
  - (1837s, 1501)
Input Tuning

- **10-minute coverage (input tuning) $\geq$ 1-hour coverage (other methods)**

- SUSY-HMC: 1-hour coverage (input tuning) is about 1.2-2.3X higher than 1-hour coverage (other methods)
Floating-point Extension

![Graph showing the comparison of Real, Int, and Float branches over time. The graph illustrates the number of branches against time (s). The Real branch has the highest number of branches, followed by the Int and then the Float branch.]
Floating-point Extension

- Real (1704) > Int (1662) > Float (1582)

- Constraint solving time of Real (1.7%) < Constraint solving time of Float (10.9%)
Conclusion

- Input tuning
  - **10-minute** coverage (input tuning) ≥ **1-hour** coverage (other methods)
  - SUSY-HMC: 1-hour coverage (input tuning) is about **1.2-2.3X** higher than 1-hour coverage (other methods)

- Floating-point Extension
  - Floating-point extension using reals achieve 42-122 more branches than the other two
Thank you!