Test Data Generation for Exposing Interference Bugs in Multi-Threaded Systems

Neelesh Bhattacharya

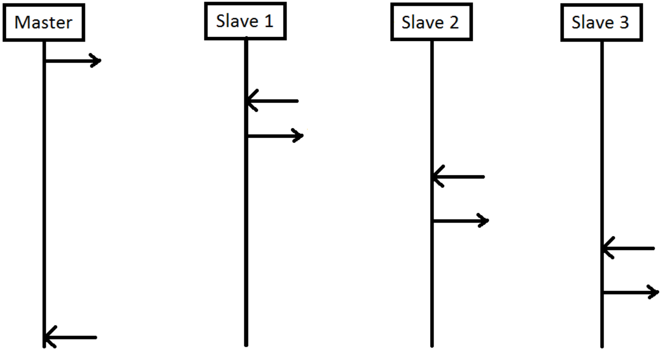
Supervisors: Giuliano Antoniol and Yann-Gaël Guéhéneuc

SOCCER Lab and Ptidej Team, DGIGL
École Polytechnique de Montréal, Québec, Canada

Thesis Presentation - December 10th, 2012

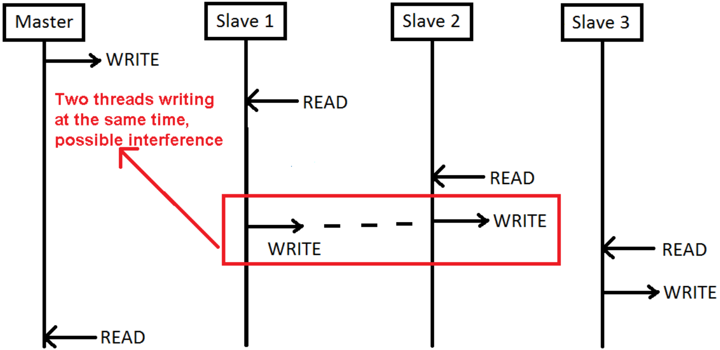
Outline

Context and Motivation (1/5)



Multi-Threading

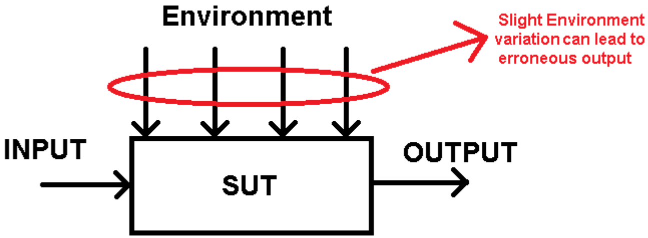
Context and Motivation (2/5)



Interference Bugs

Context and Motivation (3/5)

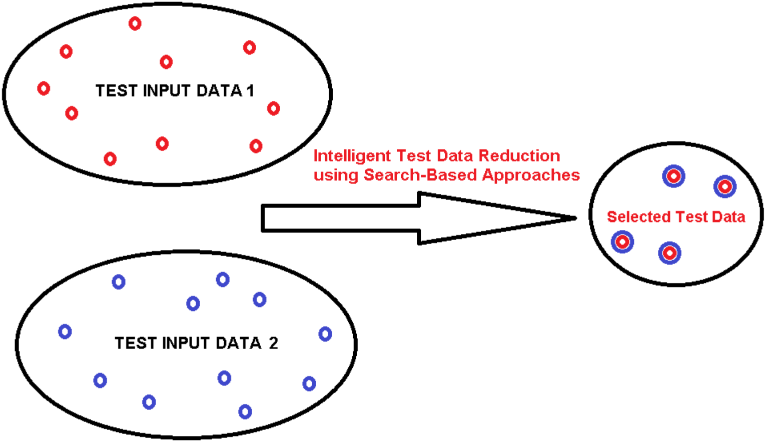
- Test data generation is a solution to expose the data-race and interference bugs



Non-Deterministic Multi-threaded Environment

Context and Motivation (4/5)

- Search-based approaches have been used to generate test data over the years



Context and Motivation (5/5)

- Use of search-based approaches in multi-threaded systems for exposing interference bugs have been limited
- Previous approaches [1] required lot of effort to expose interference bugs and did not mitigate non-determinism factor of the environment

Need to reduce the effort for finding interference bugs and mitigate the non-determinism of the multi-threaded environment

Thesis

Thesis Statement

Search-based techniques can be used effectively to generate test data to expose interference condition in multi-threaded systems

Major challenges of Search-Based Approaches

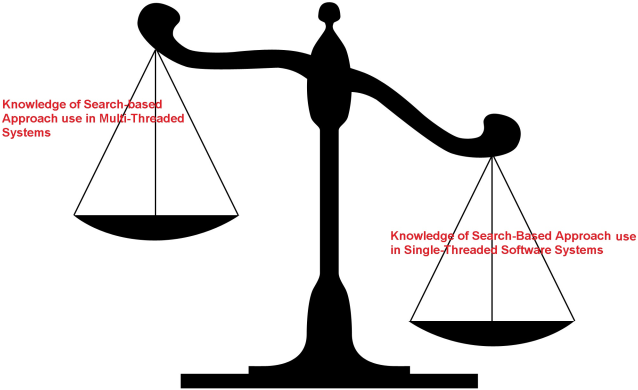
From the literature study of search-based approaches we found three major challenges:

- C1: Formulating the original problem as a search problem (R. Saravanan et Vijayakumar (2003))
- C2: Developing the right fitness function for the problem formulation (Lim et al. (2006))
- C3: Finding the right (scalable) search-based approach using scalability analysis (Fang et al. (2002))

Studies Performed (1/4)

- Use of search-based approaches for single-threaded systems have been extensive
- The use have been limited for multi-threaded systems

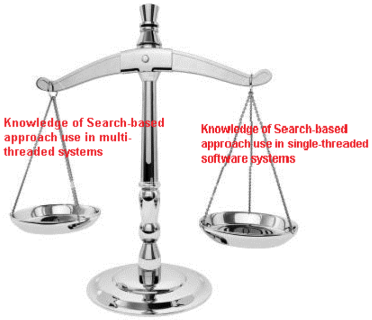
Studies Performed (2/4)



Studies Performed (3/4)

- We perform a preliminary study on single-threaded software systems to learn more about the challenges
- Study 1: Exposing divide-by-zero exceptions in single-threaded systems while addressing C1, C2 and C3

Studies Performed (4/4)



- We apply our knowledge for study 1 into the major study with multi-threaded systems
- Study 2: Exposing interference bugs in multi-threaded systems while addressing C1, C2 and C3

Study 1: Raising Divide-by-Zero Exception

Motivation

```
int z, x;
if (x<10)
        if (x>3)
                x = 1/(z-1);
else
        x = 4;
```

Motivation

```
int z, x;
if (x<10)
    if (x>3)
        x = 1/(z-1);
else
    x = 4;
```

$\boxed{\text{If } x = 5,\ z = 1}$

Motivation

```
int z, x;
if (x<10)
        if (x>3)
                x = 1/(z-1);
else
        x = 4;
```

Exception Raised, Program Failure

Related Work (1/3)

- Miller and Spooner (1976) introduced local search to generate input data to cover specific paths
- Baresel (2000) proposed the concept of approach level to enrich fitness functions
- Xanthakis et al. (1992) were the first to use Genetic Algorithm for test data generation for branch coverage

Related Work (2/3)

- Jones et al. (1976) used evolutionary based software testing approaches like genetic algorithm to generate test data automatically
- Mcminn (2004) presented a survey of evolutionary and other related techniques, emphasizing on their pros and cons
- Tracey et al. (2000) proposed an approach to automatically generate test data for raising divide-by-zero exceptions

Related Work (3/3)

Study 1 stems from the work of Tracey et al. (2000) on test data generation for raising divide-by-zero exceptions

+

- C1: They formulated branch coverage problem as a search problem

-

- C2: They proposed a fitness function which was not well guided and behaved like random search
- C3: They did not provide a scalability analysis of the search-based approaches

Problem Statement

Raising divide-by-zero exceptions for Programs Under Test (PUTs), using a set of optimization algorithms to overcome the previous limitations

C1: Reformulating the Problem (1/2)

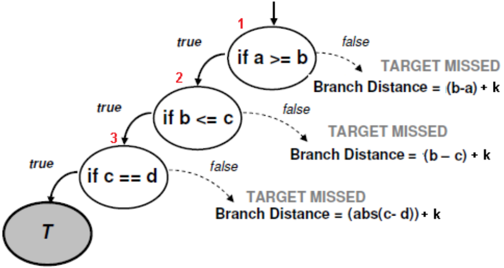
```
int z, x=4;
if (Z>1 AND Z<=5)
        return z;
else
        return (x*4)/(z-1);
```

C1: Reformulating the Problem (2/2)

```
1      int z, x=4;
2      if (Z>1 AND Z<=5)
3             return z;
4      else
5.1           if (Z == 1)
5.2                  print "Exception raised";
5.3           else
5.4                  return (x*4)/(z-1);
```

C2: Reformulating the Fitness Function (1/2)

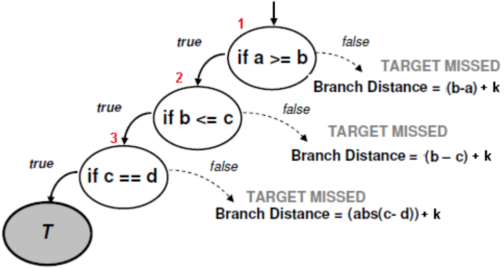
```
CFG Node
 (s)     void example(int a, int b, int c, int d)
         {
 (1)       if (a >= b)
           {
 (2)         if (b <= c)
             {
 (3)           if (c == d)
               {
                 // T
```



C2: Reformulating the Fitness Function (1/2)

Tracey Fitness Function = Branch Distance

```
CFG Node
  (s)     void example(int a, int b, int c, int d)
          {
  (1)       if (a >= b)
            {
  (2)         if (b <= c)
              {
  (3)           if (c == d)
                {
                  // T
```



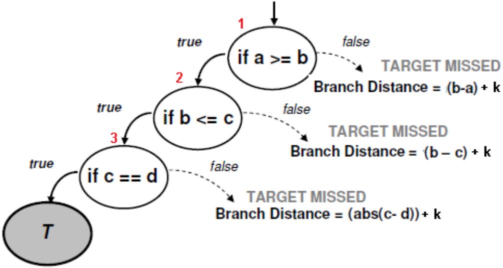
C2: Reformulating the Fitness Function (1/2)

Tracey Fitness Function = Branch Distance

```
CFG Node
  (s)     void example(int a, int b, int c, int d)
          {
  (1)       if (a >= b)
            {
  (2)         if (b <= c)
              {
  (3)           if (c == d)
                {
                  // T
```

For a = 10, b = 15, c = 20, d = 30, k = 5

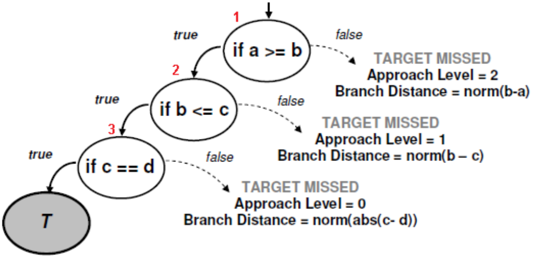
Leaving Branch 1, Fitness = 10
Leaving Branch 2, Fitness = 0
Leaving Branch 3, Fitness = 15



C2: Reformulating the Fitness Function (2/2)

CFG Node

```
(s)     void example(int a, int b, int c, int d)
        {
(1)       if (a >= b)
          {
(2)         if (b <= c)
            {
(3)           if (c == d)
              {
                // T
```

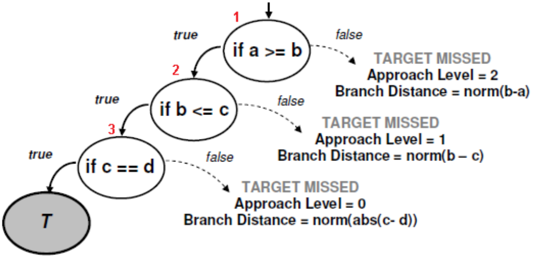


C2: Reformulating the Fitness Function (2/2)

$$\text{Fitness Function} = \text{Approach Level} + (1 - 1.001^{-\text{branch distance}})$$

CFG Node

```
(s)     void example(int a, int b, int c, int d)
        {
(1)       if (a >= b)
          {
(2)         if (b <= c)
            {
(3)           if (c == d)
              {
                // T
```



C2: Reformulating the Fitness Function (2/2)

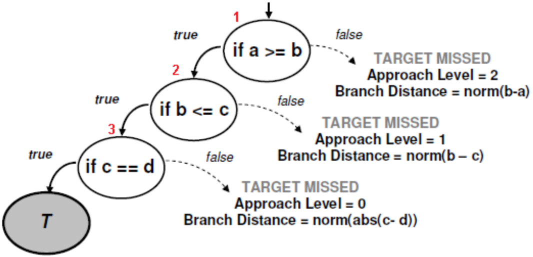
Fitness Function = Approach Level + $(1 - 1.001^{\text{-branch distance}})$

CFG Node

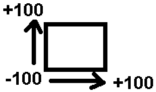
```
(s)     void example(int a, int b, int c, int d)
        {
(1)       if (a >= b)
          {
(2)         if (b <= c)
            {
(3)           if (c == d)
              {
                // T
```

For a = 10, b = 15, c = 20, d = 30, k = 5

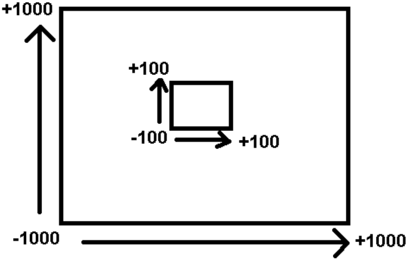
Leaving Branch 1, Fitness ~=3
Leaving Branch 2, Fitness = 1+1 = 2
Leaving Branch 3, Fitness~=1



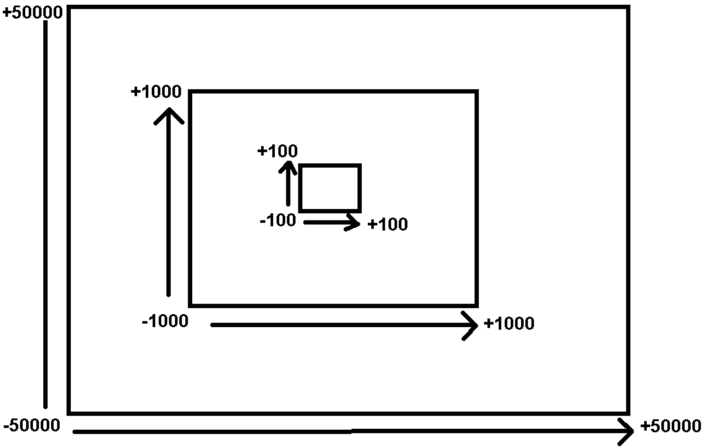
C3: Scalability Analysis



C3: Scalability Analysis



C3: Scalability Analysis



Research Questions

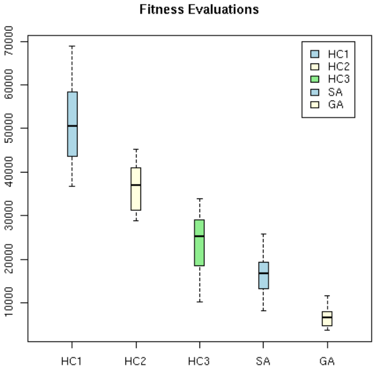
Which of Tracey's fitness function and our fitness function is more effective in raising divide-by-zero exception?

- **RQ1**: Which is the best hill climbing variant?
- **RQ2**: Which is the most effective divide-by-zero exception raising technique?
- **RQ3**: Which of Tracey's fitness function and our fitness function is more efficient?
- **RQ4**: Which among CP and metaheuristics is better for divide-by-zero exception raising?

Systems Studied

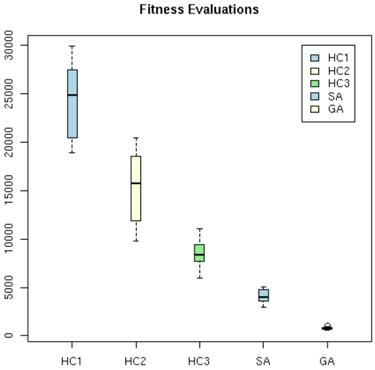
- Tracey exemplary code
- GridCanvas Class from Eclipse 2.0.1
- ProcessStats Class from Android 2.0

HC3 best HC variant, GA best approach (Tracey Code)



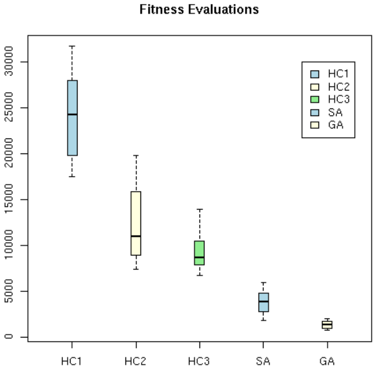
RQ1 and RQ2

HC3 best HC variant, GA best approach (Eclipse UUT)



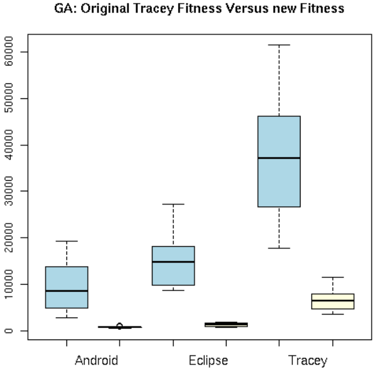
RQ1 and RQ2

HC3 best HC variant, GA best approach (Android UUT)



RQ1 and RQ2

Proposed Fitness function performs better



RQ3

Results are statistically significant (Tracey Code)

Comparisons	*p*-values	Cohen *d* values
HC1-HC2	9.261e-10	2.55246
HC1-HC3	6.376e-16	5.951003
HC2-HC3	6.868e-08	2.428475
HC3-SA	7.049e-14	4.147889
HC3-GA	2.2e-16	8.223645
SA-GA	8.763e-15	6.254793

Results are statistically significant (Eclipse UUT)

Comparisons	*p*-values	Cohen *d* values
HC1-HC2	3.245e-10	2.682195
HC1-HC3	7.167e-13	4.111778
HC2-HC3	0.003998	0.9912142
HC3-SA	2.387e-11	3.239916
HC3-GA	1.933e-13	5.258295
SA-GA	9.989e-09	2.694495

Results are statistically significant (Android UUT)

Comparisons	*p*-values	Cohen *d* values
HC1-HC2	1.438e-06	1.894204
HC1-HC3	2.981e-12	3.345377
HC2-HC3	7.438e-08	2.12037
HC3-SA	0.0003169	1.266088
HC3-GA	1.283e-10	3.481401
SA-GA	4.531e-09	2.696728

Execution Time: CP is superior

	Tracey Exemplary Code	Eclipse	Android
GA	8.067/1.439	2.129/1.149	1.926/1.177
CP	1.035/0.0135	0.01/0	0.01/0

RQ4

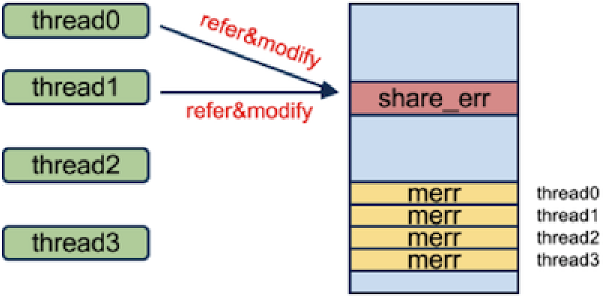
Results are statistically significant

Comparisons	*p*-values	Cohen *d* values
HC1-HC2	1.438e-06	1.894204
HC1-HC3	2.981e-12	3.345377
HC2-HC3	7.438e-08	2.12037
HC3-SA	0.0003169	1.266088
HC3-GA	1.283e-10	3.481401
SA-GA	4.531e-09	2.696728

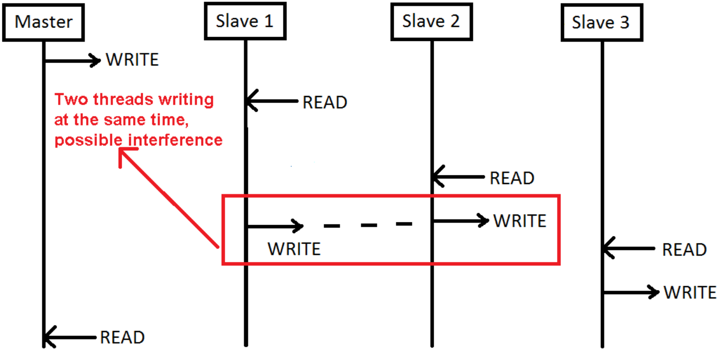
Contributions

- C1: Reformulate existing divide-by-zero exception problem as branch coverage problem
- C2: Reformulate the fitness function to the branch coverage problem
- C3: Analyse the scalability of HC, SA, GA, RND and compare them to CP

Study 2: Exposing Interference Bugs



Motivation



Interference Bugs

Related Work (1/3)

- Artho et al. (2003) provided a higher abstraction level for data races to detect inconsistent uses of shared variables
- Hovemeyer and Pugh (2004) used bug pattern detectors to find correctness and performance-related bugs in several Java programs
- Bradbury et al. (2006) proposed a set of mutation operators for concurrency programs used to mutate the portions of code

Related Work (2/3)

- Lei et Carver (2006) used reachability testing to generate synchronization sequences automatically and on-the-fly
- ConTest (Edelstein et al. (2003)), a lightweight testing tool used various heuristics to create scheduling variance in multi-threaded systems
- CHESS (Mutuvasi et al. (2007)), a concurrent systems testing tool tries all the possible schedules to find a bug

Related Work (3/3)

+

- Previous approaches found interference bugs by testing all possible schedules

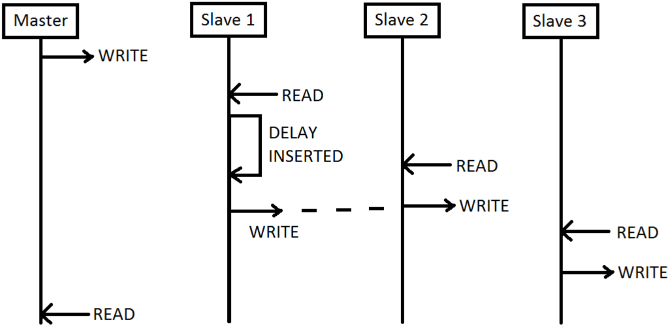
-

- Do not mitigate the non-determinism of the multi-threaded environment
- Requires lot of effort (time and testing resources) to find a bug

Problem Statement

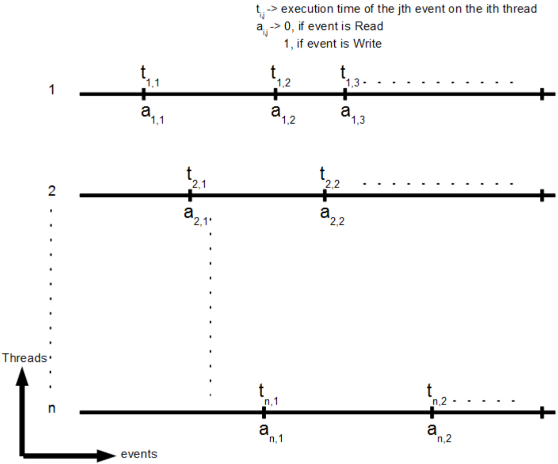
Expose interference bug pattern for Programs Under Test (PUTs), using a set of optimization algorithms to mitigate the limitations

C1: Reformulating the Problem (2/2)



Single delay injected

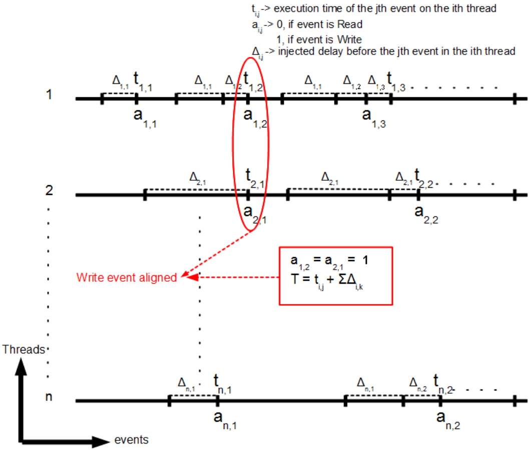
C2: Reformulating the Fitness Function (1/2)



Test Data Generation for Exposing Interference Bugs in Multi-Threaded Systems

Neelesh Bhattacharya

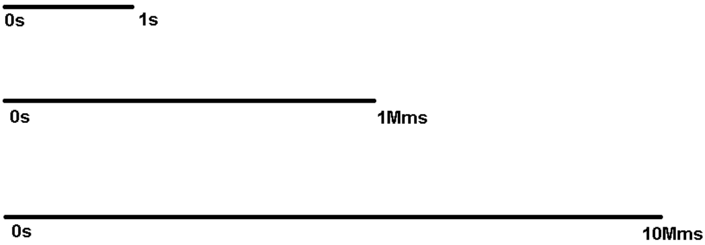
C2: Reformulating the Fitness Function (2/2)



C3: Scalability Analysis

C3: Scalability Analysis

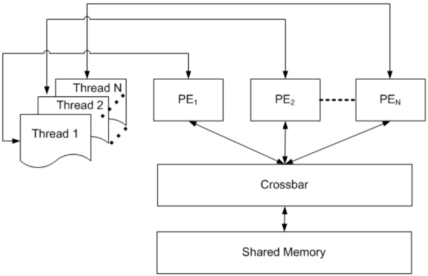
C3: Scalability Analysis



Research Questions

- How to maximize interference probability between threads?
- How the search space dimension impacts the search algorithm performance?

ReSP: The Virtual Platform



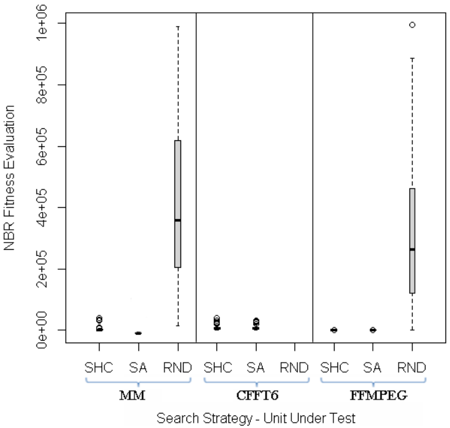
- No effect of the OS
- No delay in the hardware
- No cache effects

Thus our focus is only on the bug inherent in the code

Systems Studied

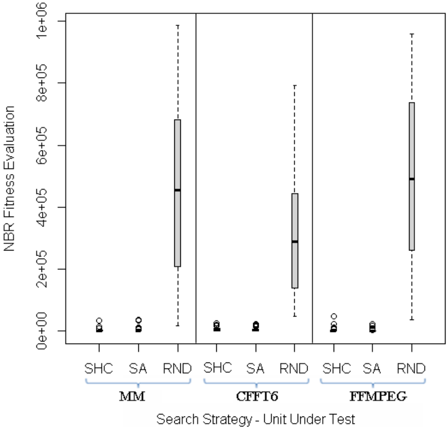
- Matrix Multiplication (MM)
- Count Shared Data (CSD)
- Average of Numbers (AvN)
- Area of Circle (AC)
- CFFT
- CFFT6
- FFMPEG

SA, SHC outperforms RND



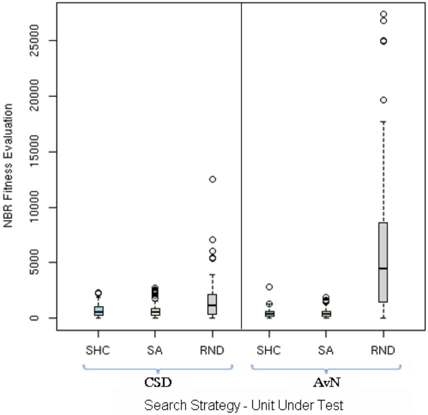
Search space upto 10Mms delay **(RQ1)**

SA, SHC outperforms RND



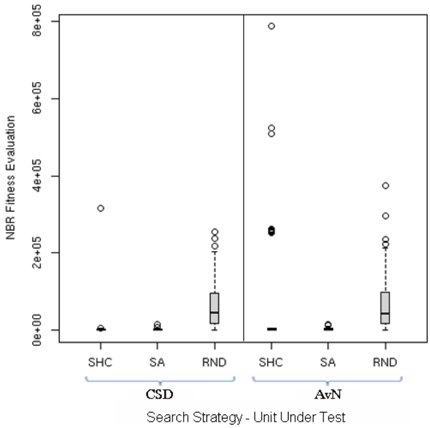
Search space upto 1Mms delay **(RQ2)**

SA, SHC outperforms RND



Search space upto 10s delay **(RQ2)**

SA, SHC outperforms RND



Search space upto 1s delay **(RQ2)**

Execution Time: SA, SHC outperforms RND

Execution Times for Real-World Applications in ms

	CFFT		CFFT6		FFMPEG	
	1×10^6	1×10^7	1×10^6	1×10^7	1×10^6	1×10^7
SA	3118	5224	27443	2041	1562	4672
HC	3578	4976	27328	21943	1378	5100
RND	113521	107586	342523	339951	59599	133345

Contributions

- C1: Formulate the interference bugs exposing problem as a delay-injection search problem
- C2: Develop a novel fitness function based on the problem formulation
- C3: Analyse the scalability of SHC, SA and RND
- Use ReSP for having deterministic environment

Threats to Validity (1/2)

Study 1

- Construct Validity: One of the studied systems was a synthetic one
- Internal Validity: We limited the bias of intrinsic randomness of our results by repeating each experiment 20 times
- Conclusion Validity: We inspected box-plots, performed t-tests, and evaluated the Cohen d effect sizes
- External Validity: Evaluation was conducted on 3 systems, larger evaluation is desirable

Threats to Validity (2/2)

Study 2

- Construct Validity: Four of the studied systems were synthetic ones
- Internal Validity: Three of the four synthetic systems were developed long before the analysis, thus are unbiased
- External Validity: Evaluation was conducted on 7 systems, larger evaluation is desirable

Conclusion (1/3)

- We propose an approach to see the effectiveness of search-based approaches in exposing interference bugs
- We found three challenges C1, C2 and C3 of using search based approaches from related work
- Search based approaches have been used extensively in single-threaded systems, unlike multi-threaded systems

Conclusion (2/3)

- We use the knowledge of search based approach use in single-threaded systems to be applied in multi-threaded systems
- We conduct a empirical study to see the effect of using search based approaches in single-threaded software systems for raising divide-by-zero exception

Conclusion (3/3)

- Using the knowledge of study 1, we conduct our major study of using search based approaches for exposing interference bugs, in the context of C1, C2 and C3
- From both the studies involving ten systems, we find that search based approaches can be effectively used to expose interference bugs in multi-threaded systems

Future Work (1/2)

Study 1

- We would generalize our study by validating our approach with more real-world systems which are very complex
- We would extend our approach to other kinds of exceptions like null-pointer and buffer overflow
- We would raising exceptions with codes having complex data structures

Future Work (2/2)

Study 2

- We would validate our approach by running the systems on real hardware platform in presence of non-deterministic environment
- We would generalize our approach further by using more real-world, complex systems as case study
- We would propose extensions to our approach for exposing other bug patterns like deadlock

Summary

Thesis

Search-based techniques can be effective in exposing interference bugs

Contributions

Study 1

- C1: Reformulate divide-by-zero exception problem
- C2: Reformulate fitness function accordingly
- C3: Analyse the scalability of HC, SA, GA, RND, CP

Study 2

- C1: Formulate the interference bugs exposing problem
- C2: Develop a novel fitness function accordingly
- C3: Analyse the scalability of SHC, SA and RND
- Use ReSP to have deterministic environment