Search-Based Compiler Testing and Debugging

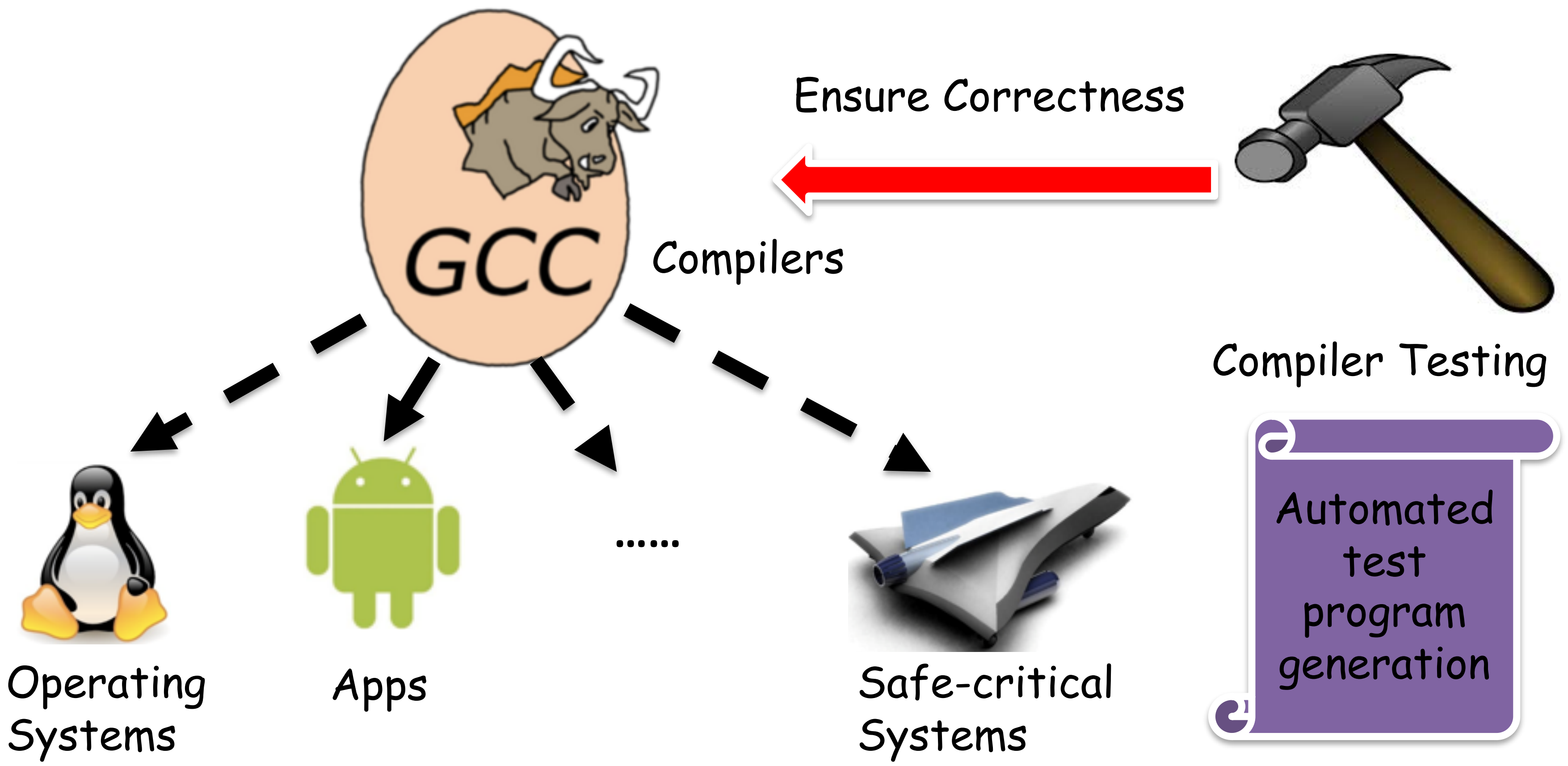
Lu Zhang

November 17, 2018

Agenda

- A Search-Based Technique for Compiler Test Generation
- A Search-Based Technique for Compiler Debugging

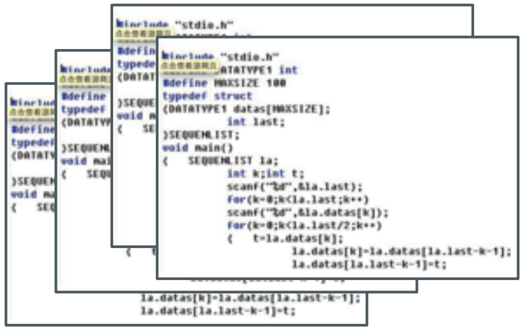
Compiler Testing



Test Program Generation

Csmith

A test configuration
(consisting of many options)



```
#include "stdio.h"
#define DATATYPE1 int
#define MAXSIZE 100
typedef struct
{DATATYPE1 datas[MAXSIZE];
    int last;
}SEQUENLIST;
void main()
{   SEQUENLIST la;
     int k;int t;
     scanf("%d",&la.last);
     for(k=0;k<la.last;k++)
     scanf("%d",&la.datas[k]);
     for(k=0;k<la.last/2;k++)
     {   t=la.datas[k];
          la.datas[k]=la.datas[la.last-k-1];
          la.datas[la.last-k-1]=t;
```

Each option directly reflects the probability of a specific program feature to be included.

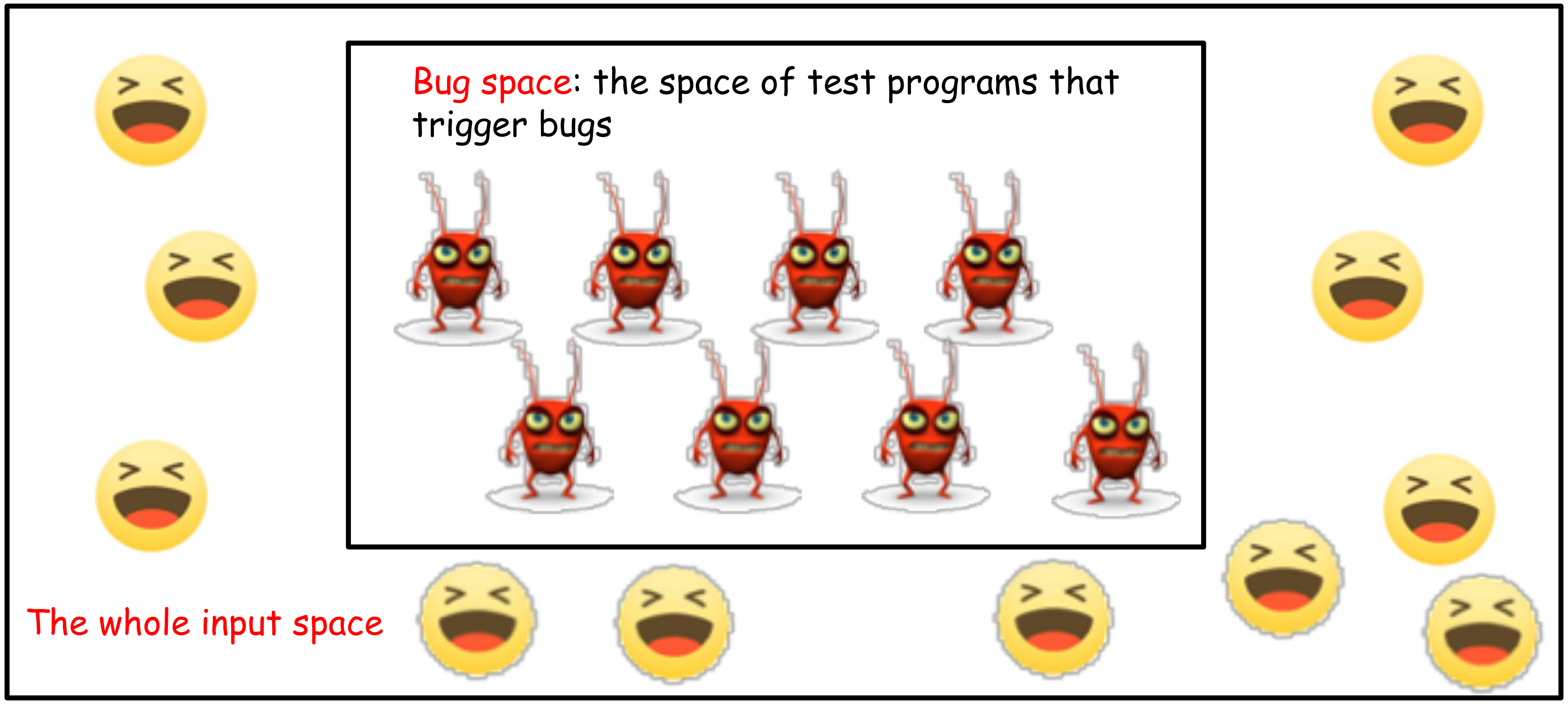
Find bugs as many as possible

Ideal goal

- Challenge 1: It is more important to generate test programs that are more likely to trigger bugs
  - what configuration would lead to such test programs
- Challenge 2: It is important to improve the diversity of the generated test programs to cover a wide range of compiler bugs
  - swarm testing, randomizing the configuration options could lead to more bugs being discovered

Our solution: *to find a set of bug-revealing and diverse test configurations*

Approach



- Criteria 1: Each test configuration in the desired set should be able to generate test programs exploring a (large) portion of bug space
- Criteria 2: The set of test configurations should have diversity for bug detection

History-based Range Inferring

$$\begin{tikzcd}
\text{option} \arrow[r, shift left, "\text{Control generation}"] & \text{Program feature} \arrow[l, shift left, "\text{Infer range}"]
\end{tikzcd}$$

PF = {pf1,pf2,…,pfm}
PP = {pp1,pp2,…,ppn}
p = {e1,e2,…,er}
c = {o1,o2,…,or}

$$Pr(i) = \frac{\sum_{j=1}^{N} e_{ij}}{\sum_{j=1}^{N} t_{ij}}$$

History-based Range Inferring

Emerging patterns: item sets whose supports change significantly between the two datasets
Big support difference: using the support of an item set on one dataset to subtract the support of the item set on another dataset

The differences reflect the range where failing test programs are easier to generate while passing test programs are more difficult to generate to some degree

$$R(o_i) = \begin{cases} [d_i + \mathit{diff}(i), d_i], & \mathit{diff}(i) < 0 \\ [d_i, d_i + \mathit{diff}(i)], & 0 \leq \mathit{diff}(i) \leq 100 - d_i \\ [d_i, 100], & \mathit{diff}(i) > 100 - d_i \end{cases}$$

Diversity Measuring

Using the distance between these feature vectors to measure the diversity of test programs

C = {c1, c2, . . . , cg}
ci = {oi1, oi2, . . . , oir}
Pi = {pi1,pi2,…,pis}

Intuitively, the generated test programs under a test configuration tend to concentrate on an area of input space.

- HDTest first sets a group center for these generated test programs, and then computes the distance between different group centers.
- Manhattan distance

$$Diversity(c_i) = min_{j\in[1,g]\&j\neq i}(Dist(pc_i, pc_j))$$

PSO-based Searching

- Expected output: a set of diverse test configurations exploring the whole bug space
- Search space: inferred range for each option
- Fitness function: diversity

$$v_{ij}^{t+1} = \omega v_{ij}^{t} + \varsigma_1 \gamma_1 (b_{ij}^{t} - c_{ij}^{t}) + \varsigma_2 \gamma_2 (b_{gj}^{t} - c_{ij}^{t})$$

$$c_{ij}^{t+1} = c_{ij}^{t} + v_{ij}^{t+1}$$

- After producing a set of bug-revealing and diverse test configurations, HDTest randomly selects a test configuration from the set to generate a test program.

Evaluation

- RQ1: How does HDTest perform compared with existing compiler test-program generation approaches?
- RQ2: Does HDTest perform well in different scenarios (including cross-version and cross-compiler scenarios)?
- RQ3: Does HDTest perform well for the latest release compiler version?

$$\begin{tikzcd} \text{GCC-4.3.0} \arrow[r] & \begin{array}{l}\text{GCC-4.4.0}\\ \text{GCC-4.5.0}\\ \text{GCC-4.6.0}\\ \text{LLVM-2.6}\\ \text{LLVM-6.0.1}\end{array} \end{tikzcd}$$

Compared approaches: DefaultTest & SwarmTest

Number of detected bugs

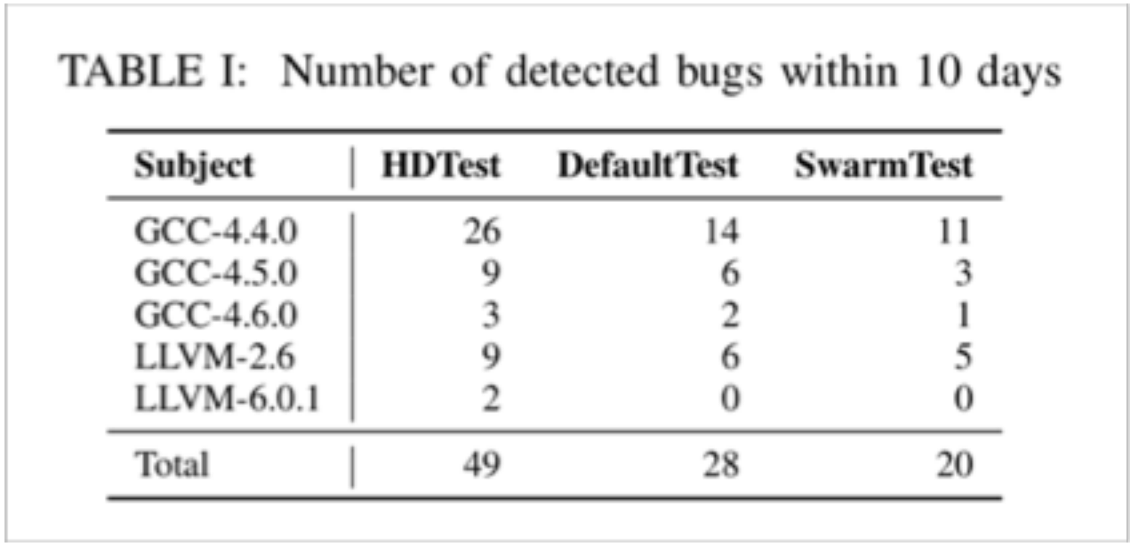
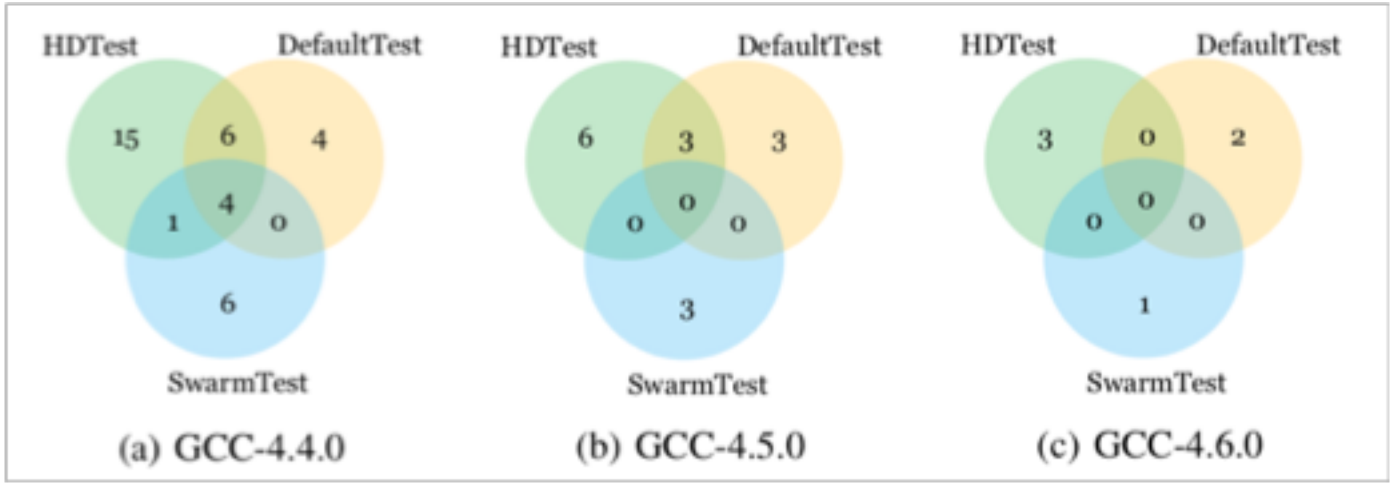


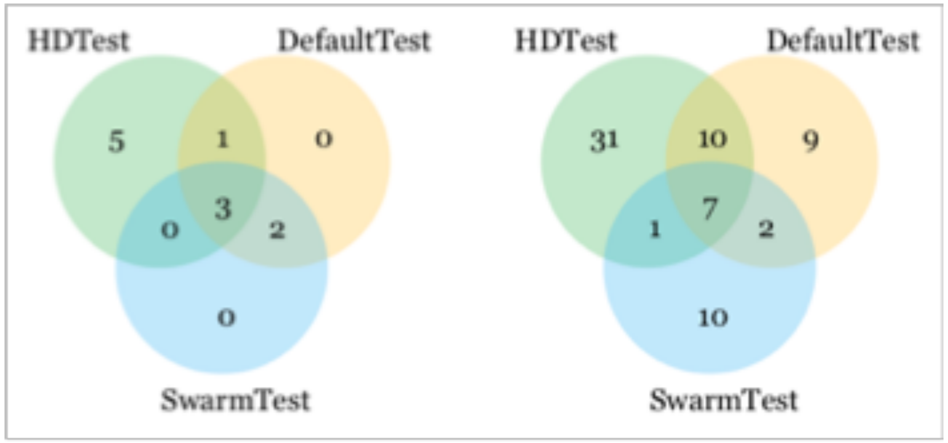
TABLE I: Number of detected bugs within 10 days

Subject	HDTest	DefaultTest	SwarmTest
GCC-4.4.0	26	14	11
GCC-4.5.0	9	6	3
GCC-4.6.0	3	2	1
LLVM-2.6	9	6	5
LLVM-6.0.1	2	0	0
Total	49	28	20

Achieving 75.00% and 145.00% improvements compared with DefaultTest and SwarmTest

Number of unique bugs





HDTest: 63.26% (31 out of 49)
DefaultTest: 32.14% (9 out of 28)
SwarmTest: 50.00% (10 out of 20)

Time spent on detecting each bug

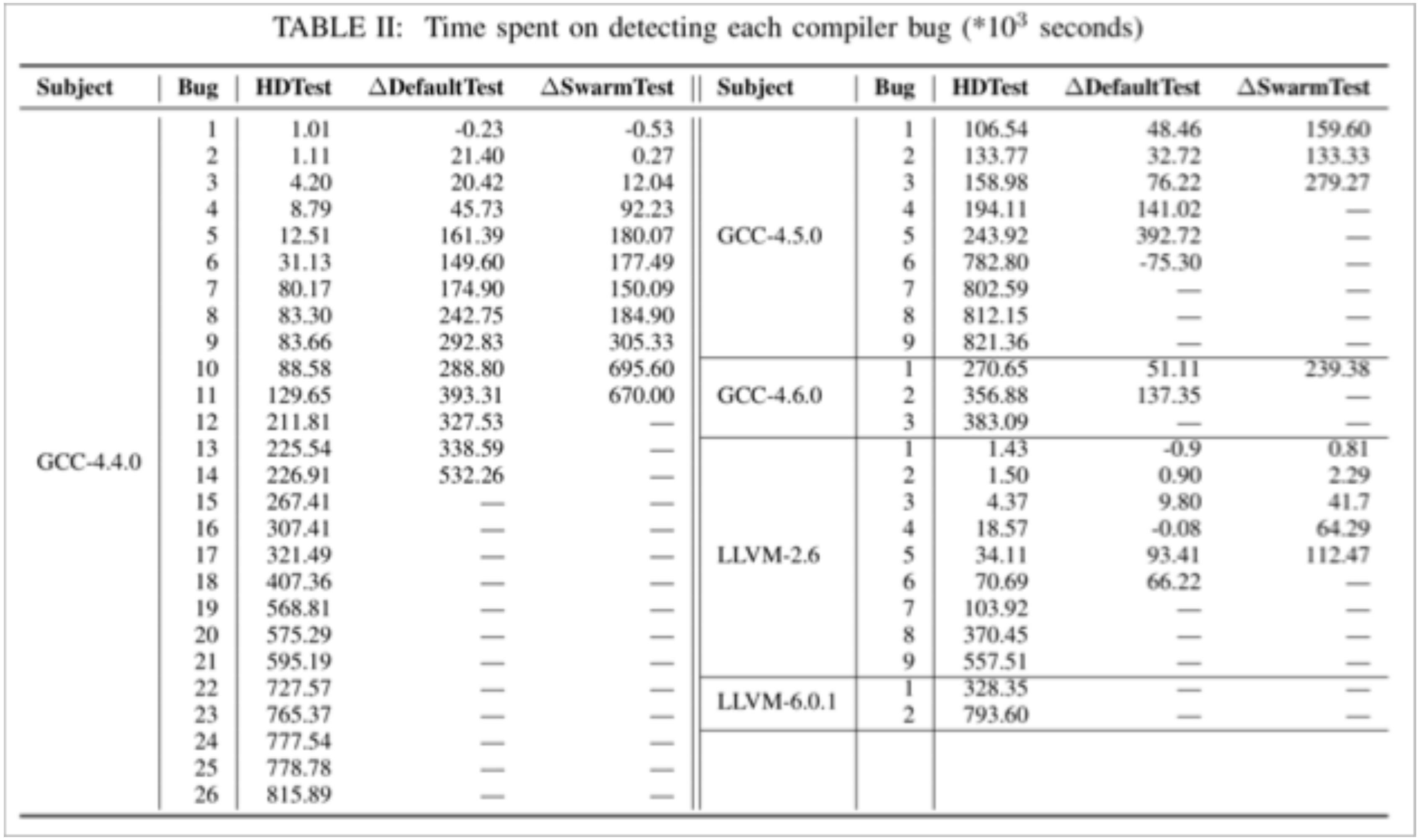


TABLE II: Time spent on detecting each compiler bug (*10^3 seconds)

Subject	Bug	HDTest	ΔDefaultTest	ΔSwarmTest	Subject	Bug	HDTest	ΔDefaultTest	ΔSwarmTest
	1	1.01	-0.23	-0.53		1	106.54	48.46	159.60
	2	1.11	21.40	0.27		2	133.77	32.72	133.33
	3	4.20	20.42	12.04		3	158.98	76.22	279.27
	4	8.79	45.73	92.23		4	194.11	141.02	—
	5	12.51	161.39	180.07	GCC-4.5.0	5	243.92	392.72	—
	6	31.13	149.60	177.49		6	782.80	-75.30	—
	7	80.17	174.90	150.09		7	802.59	—	—
	8	83.30	242.75	184.90		8	812.15	—	—
	9	83.66	292.83	305.33		9	821.36	—	—
	10	88.58	288.80	695.60		1	270.65	51.11	239.38
	11	129.65	393.31	670.00	GCC-4.6.0	2	356.88	137.35	—
	12	211.81	327.53	—		3	383.09	—	—
GCC-4.4.0	13	225.54	338.59	—		1	1.43	-0.9	0.81
	14	226.91	532.26	—		2	1.50	0.90	2.29
	15	267.41	—	—		3	4.37	9.80	41.7
	16	307.41	—	—		4	18.57	-0.08	64.29
	17	321.49	—	—	LLVM-2.6	5	34.11	93.41	112.47
	18	407.36	—	—		6	70.69	66.22	—
	19	568.81	—	—		7	103.92	—	—
	20	575.29	—	—		8	370.45	—	—
	21	595.19	—	—		9	557.51	—	—
	22	727.57	—	—	LLVM-6.0.1	1	328.35	—	—
	23	765.37	—	—		2	793.60	—	—
	24	777.54	—	—					
	25	778.78	—	—					
	26	815.89	—	—					

Speedup

$$Speedup(i) = \frac{T_{Comp}(i) - T_{HDTest}(i)}{T_{Comp}(i)}$$

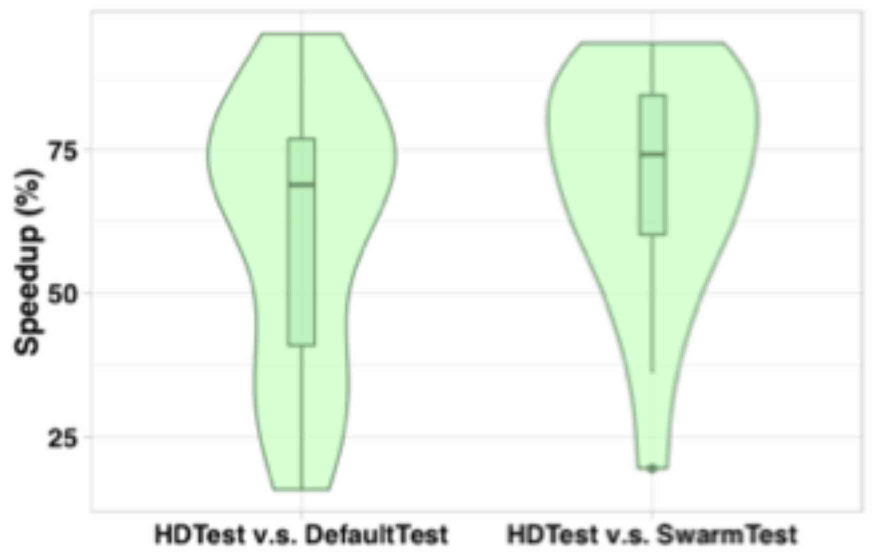


Fig. 2: Speedup distribution

Median speedups of HDTest compared with DefaultTest and SwarmTest are 68.86% and 74.14%

Debugging Compilers

- Compiler bugs are very difficult to debug
  - Most compiler bugs are deep bugs
    - E.g., optimization bugs
  - Compilers are large
    - Compiling a program involves a lot of compiler files
  - Compilation is time-consuming
  - Manual debugging tools are rarely useful

Basic Idea

- Spectrum-based fault localization
  - Using a set of passing tests (i.e., witness tests) to remove suspicion of various parts in the compiler
  - Not affordable to have a large set of witness tests
    - Using only high-quality witness tests

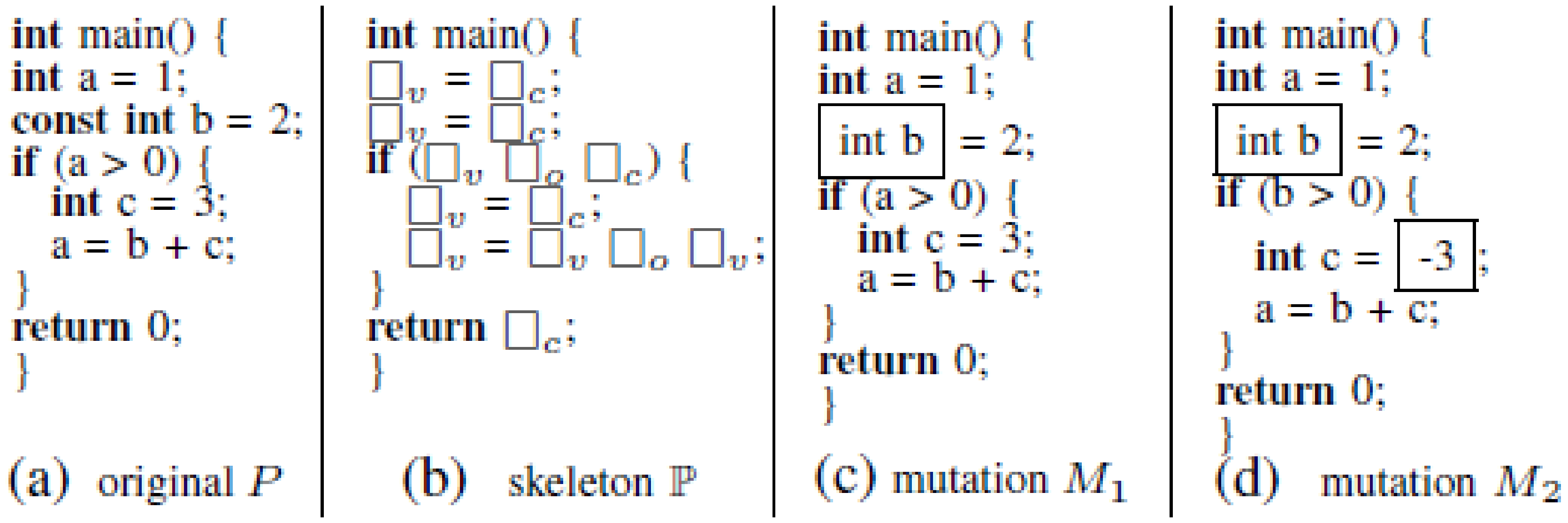
Criteria for Good Witness Tests

- Each witness test should share a similar compiler execution trace with the failing test
- Witness tests should differ much from each other in their compiler execution traces

Marple (Our Approach)

- Mutating the failing test program to obtain the witness test programs
  - Skeletal Program Mutation
- Using an adaptive process to control the quality of the witness test programs

Skeletal Program Mutation



```
int main() {
int a = 1;
const int b = 2;
if (a > 0) {
   int c = 3;
   a = b + c;
}
return 0;
}
```

(a) original P

$\text{int main() \{}$
$\square_v = \square_c;$
$\square_v = \square_c;$
$\text{if } (\square_v\ \square_o\ \square_c)\ \{$
$\quad \square_v = \square_c;$
$\quad \square_v = \square_v\ \square_o\ \square_v;$
$\}$
$\text{return } \square_c;$
$\}$

(b) skeleton $\mathbb{P}$

```
int main() {
int a = 1;
int b = 2;
if (a > 0) {
   int c = 3;
   a = b + c;
}
return 0;
}
```

(c) mutation M_1

```
int main() {
int a = 1;
int b = 2;
if (b > 0) {
   int c = -3;
   a = b + c;
}
return 0;
}
```

(d) mutation M_2

Seed Program Selection

- From first order mutations to high order mutations
  - The higher order, the more different from the failing test program
- Lower order mutations having priority
  - Aiming to have similar compiler execution traces with the failing test

How to obtain passing test programs with low cost?

- Relying more on mutation rules with better performance
  - Markov Chain Monte Carlo Methods
    - Recording the previous performance
    - Rules with better performance having better chance of being selected

$$A = min\left(1, \frac{Pr(MR_b)}{Pr(MR_a)}\right) = min\left(1, (1-p)^{k_b - k_a}\right)$$

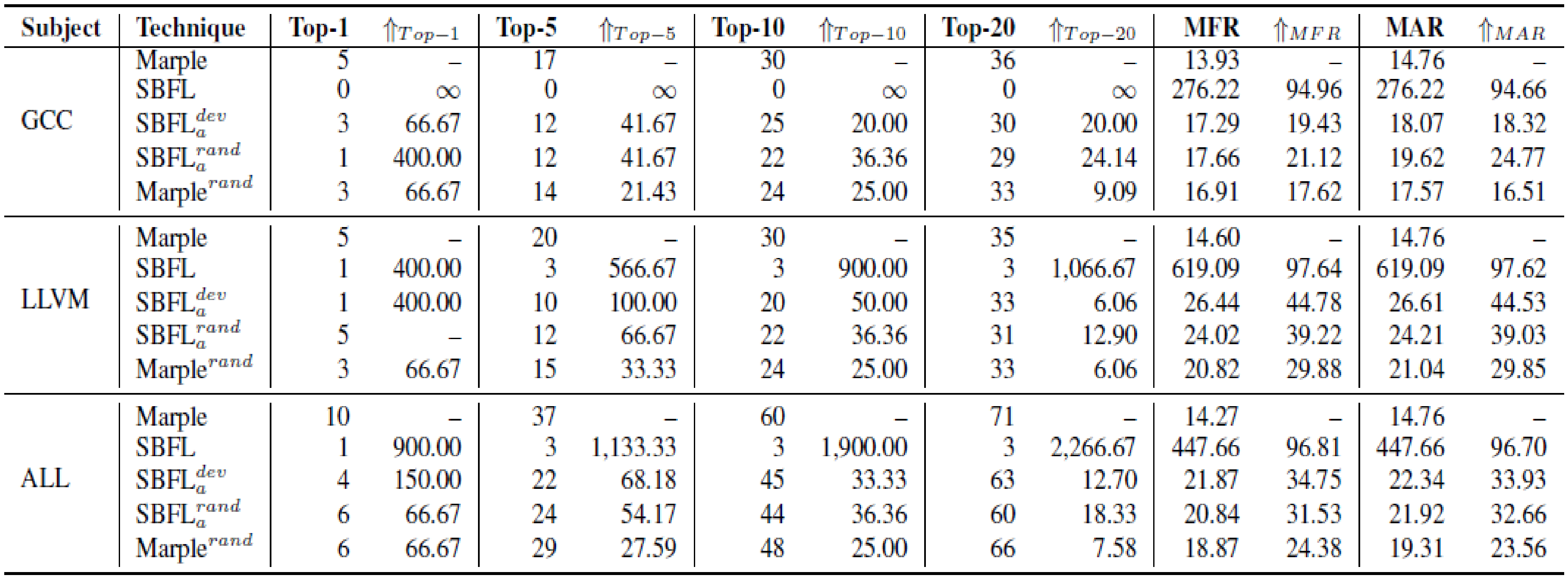
Aggregation-Based Suspicion Calculation

- Calculating suspicion values at a finer level
- Summing up the suspicion values at the finer level to form the suspicion values at a coarser level

Evaluation

- 45 GCC bugs and 45 LLVM bugs
  - Corresponding to 45 buggy GCC versions and 45 buggy LLVM versions
- On Average
  - A GCC buggy version has 1,588 files with 1,414K LOC
  - An LLVM buggy version has 3,507 files with 1,431K LOC

Results



Subject	Technique	Top-1	$\Uparrow_{Top-1}$	Top-5	$\Uparrow_{Top-5}$	Top-10	$\Uparrow_{Top-10}$	Top-20	$\Uparrow_{Top-20}$	MFR	$\Uparrow_{MFR}$	MAR	$\Uparrow_{MAR}$
GCC	Marple	5	–	17	–	30	–	36	–	13.93	–	14.76	–
	SBFL	0	∞	0	∞	0	∞	0	∞	276.22	94.96	276.22	94.66
	SBFL$_a^{dev}$	3	66.67	12	41.67	25	20.00	30	20.00	17.29	19.43	18.07	18.32
	SBFL$_a^{rand}$	1	400.00	12	41.67	22	36.36	29	24.14	17.66	21.12	19.62	24.77
	Marple^{rand}	3	66.67	14	21.43	24	25.00	33	9.09	16.91	17.62	17.57	16.51
LLVM	Marple	5	–	20	–	30	–	35	–	14.60	–	14.76	–
	SBFL	1	400.00	3	566.67	3	900.00	3	1,066.67	619.09	97.64	619.09	97.62
	SBFL$_a^{dev}$	1	400.00	10	100.00	20	50.00	33	6.06	26.44	44.78	26.61	44.53
	SBFL$_a^{rand}$	5	–	12	66.67	22	36.36	31	12.90	24.02	39.22	24.21	39.03
	Marple^{rand}	3	66.67	15	33.33	24	25.00	33	6.06	20.82	29.88	21.04	29.85
ALL	Marple	10	–	37	–	60	–	71	–	14.27	–	14.76	–
	SBFL	1	900.00	3	1,133.33	3	1,900.00	3	2,266.67	447.66	96.81	447.66	96.70
	SBFL$_a^{dev}$	4	150.00	22	68.18	45	33.33	63	12.70	21.87	34.75	22.34	33.93
	SBFL$_a^{rand}$	6	66.67	24	54.17	44	36.36	60	18.33	20.84	31.53	21.92	32.66
	Marple^{rand}	6	66.67	29	27.59	48	25.00	66	7.58	18.87	24.38	19.31	23.56

Practicability

- A small survey with 7 compiler developers
  - Sending out 10 requests
- 6 developers confirming that compiler debugging starts with identifying the faulty file
- 6 developers considering our tool of practical value

Thank You!