The Journal of Systems and Software 104 (2015) 69–81

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Automated fault localization via hierarchical multiple predicate switching

Xiaoyan Wang, Yongmei Liu*

School of Information Science and Technology, Sun Yat-sen University, 132 East Waihuan Road, Higher Education Mega Center, Guangzhou, 510006, P.R. China

ARTICLE INFO

Article history:
Received 26 February 2014
Revised 4 January 2015
Accepted 15 February 2015
Available online 28 February 2015

Keywords:
Fault localization
Predicate
Multiple predicate switching

ABSTRACT

Single predicate switching forcibly changes the state of a predicate instance at runtime and then identifies the root cause by examining the switched predicate, called critical predicate. However, switching one predicate instance has its limitations: in our experiments, we found that single predicate switching can only find critical predicates for 88 out of 300 common bugs in five real-life utility programs. For other 212 bugs, overcoming them may require switching multiple predicate instances. Nonetheless, taking all possible combinations of predicate instances into consideration will result in exponential explosion. Therefore, we propose a hierarchical multiple predicate switching technique, called HMPS, to locate faults effectively. Specifically, HMPS restricts the search for critical predicates to the scope of highly suspect functions identified by employing spectrum-based fault localization techniques. Besides, instrumentation methods and strategies for switch combination are presented to facilitate the search for critical predicates. The empirical studies show that HMPS is able to find critical predicates for 111 out of 212 bugs mentioned above through switching multiple predicate instances. In addition, HMPS captures 62% of these 300 bugs when examining up to 1% of the executed code, while the Barinel and Ochiai approaches locate 18% and 16% respectively.

1. Introduction

A general approach to automated program debugging is based on modifying the program state to bring the program execution to a successful completion. However, searching for arbitrary state change is difficult due to the extremely large search space. A solution proposed by Zhang et al. (2006) is to switch the outcome of an instance of a predicate, and the predicate instance whose switching produces the correct result is called a *critical predicate*. The search space for predicate switching is far less than that for arbitrary state changes of other data types, like integer, because a predicate instance only has two states: *TRUE* and *FALSE*. Moreover, critical predicates are good inspecting start points for further localization and may offer useful clues for finally pinpointing the root cause, since critical predicates are data or control dependent on the root causes. Besides, the switched predicate only relates to one failed test case, hence the approach can provide values of variables and execution times of statements to developers to enhance the explanatory ability of the localization results (Parnin and Orso, 2011). Nonetheless, single predicate switching has its limitations in locating more complex faults which may require simultaneously altering states of multiple predicate instances at runtime. However, the search space for naive multiple predicate switching grows exponentially in the number of predicate instances in a run.

In this paper, we propose an efficient multiple predicate switching approach, called HMPS, to identify critical predicate sets for faulty programs. Specifically, HMPS first selects highly suspect functions via the block hit spectrum based technique (Abreu et al., 2006; Perez et al., 2014) with each block instantiated as a function. If necessary, among the selected suspicious functions, HMPS finds combinations of functions according to the ranking of these functions and the call graph of the faulty program. By doing this, the original faulty program is separated into ordered pieces. With the proposed instrumentation approach and strategies for switch combinations, HMPS then does multiple predicate switching in the highly suspect pieces one by one. Our method finally reports all statements on which the critical predicates are data or control dependent. If the correct outputs can be obtained via switching the outcomes of a set of predicate instances, this set is called a *critical predicate instance set* (CPIS), and the set of predicates of a CPIS is called a *critical predicate set* (CPS).

The main contributions of this paper include: (1) We take advantage of the block hit spectrum based technique to implement efficient multiple predicate switching. (2) To explore strategies for switch combinations readily, a source code instrumenter is developed to instrument linear switch arrays for predicates within statements. (3) Some experiments are put forward to evaluate the proposed method

* Corresponding author. Tel.: +8613802753652/8613512747729.
E-mail addresses: wangxy25@mail2.sysu.edu.cn (X. Wang), ymliu@mail.sysu.edu.cn (Y. Liu).

http://dx.doi.org/10.1016/j.jss.2015.02.038
0164-1212/© 2015 Elsevier Inc. All rights reserved.

by comparing it with Barinel (Abreu et al., 2009b) and Ochiai (Abreu et al., 2006) on 300 common bugs in five real-life programs.

The remainder of this paper is organized as follows. The next section presents two motivational examples and some statistics about predicates. Section 3 first illustrates a top level overview and the algorithm of HMPS; and then depicts the instrumentation method and the complexity of HMPS. Section 4 evaluates HMPS on 300 common bugs from five real-life utility programs. Section 5 discusses related work. Section 6 closes with conclusions and future work.

2. Motivational examples and some statistics on predicates

In this section, two bug examples are first illustrated to show the value of multiple predicate switching. Statistics about predicates and their instances in five real-life programs are then listed to explain the feasibility of implementing multiple predicate switching in functions.

2.1. Motivational examples

In the following, we present pieces of two faulty versions of the utility program grep.[1] In the code listing, `LN` represents the line number of a statement, `P` is the number of times a statement is executed in the run of the correct version of grep, and `F` is the number of times a statement is executed in the same test of a faulty version.

The following piece of code describes a bug at line 1900 in grep-2.6. In the buggy run, the value of the only instance of the Boolean expression `strncmp(s,s,len)==0` at line 1900 is 1, while the values of the 12 instances of the predicate in the correct run are all 0. There are two possible solutions for predicate switching. One solution is to change all the return values of function `looking_at()` to be 0. Coincidentally, the *TRUE* branch of the predicate at line 1898 is “`return 0;`”. Therefore, if the statement at line 1898 can be executed 12 times with all taking the *TRUE* branch, then the buggy run can be changed to output the correct result. Another solution is to flip all states of predicate instances at line 2189. We observe that function `looking_at()` is only invoked by function `lex()` at line 2189. In the buggy run, the predicate at line 2189 is executed only once by taking the *TRUE* branch. By contrast, in the correct run, the predicate at line 2189 takes the *FALSE* branch in all of its 12 instances. Accordingly, if states of all instances of the predicate at line 2189 are flipped in the buggy run, then the failed test will be fixed. In our experiment, we found that either simultaneously flipping states of 12 predicate instances at line 1898 or states of 12 predicate instances at line 2189 successfully produces the correct output.

```
 (P)  (F)   (LN)
  2:   1:   1907: lex (void){ ...
  -:   -:   1988:  do{
  9:   1:   2187:     if (c == '[' && (syntax_bits & RE_CHAR_CLASSES))
 13:   1:   2188:     for (c1 = 0; prednames[c1].name; ++c1)
 12:   1:   2189:     if (looking_at(prednames[c1].name))
                      { ...
  -:   1:   2207:       goto skip;
                      } ...
                 }}
 12:   1:   1892: static int looking_at (char const *s){ ...
 12:   1:   1897:   len = strlen(s);
 12:   1:   1898:   if (lexleft < len)
  0:   0:   1899:     return 0;
                    //correct: return strncmp(s, lexptr, len) == 0;
 12:   1:   1900:   return strncmp(s, s, len) == 0;
  -:   -:   1901: }
```

However, both the predicate at lines 1898 and 2187 only have one instance in the faulty run. How to obtain the other 11 instances for each of them? In Section 3.3, one of the instrumentation strategies will be presented to show how to extend the number of instances for predicates in such cases. As a result, the root cause is captured by our method HMPS with only examining 1.8% of the code. However, 28.9% and 31.4% of the code are needed to be inspected until the bug is located by Barinel and Ochiai respectively. In addition, there are 404,415 predicate instances in the failed execution, but only 525 and 1 predicate instances in function `lex()` and `looking_at()` respectively. As we can see, it is infeasible to take all combinations of these 404,415 predicate instances into consideration, but it may be feasible when only concentrating on predicate instances in function `lex()` or `looking_at()` via switch combination strategies.

The following piece of code describes another bug in grep-2.4.2, where there is a fault at line 3631. In the faulty run, the predicate at line 3631 takes the *TRUE* branch in 16 out of its 33 instances, while it takes the *TRUE* branch in all of its 33 instances in the correct run. Accordingly, one solution is to change the states of the other 17 instances at line 3631. It turns out that HMPS can capture the root cause with only examining 0.07% of the code, while Barinel and Ochiai need to examine 13.3% and 15.2% of the code respectively. In total, there are 34,157 predicate instances in the faulty run but only 115 in the function containing the fault. This also shows that the number of predicate instances in a function is significantly fewer than that in a program.

```
 (P)  (F)  (LN)
 33:  33:  3631:  if (result != NULL && old != NULL)
                   //correct version: (result != NULL && new != NULL)
 33:  16:  3632:    (void) strcpy(result + oldsize, new);
 33:  33:  3633:  return result;
```

```
 (P)  (F1)  (F2)  (LN)
 33:   33:  19:   3631:  if (result != NULL && old != NULL)
                         //correct version: (result != NULL && new != NULL)
 33:   16:   6:   3632:    (void) strcpy(result + oldsize, new);
 33:   33:  19:   3633:  return result;
```

The two examples above illustrate that restricting the search for critical predicate sets to functions is a feasible way to implement multiple predicate switching, especially with some switch combination strategies.

2.2. Statistics about predicates

To demonstrate the feasibility of implementing multiple predicate switching in functions, we presents some statistics on predicates and their instances in this section. We first introduce six programs for investigation:

- flex[2]—a tool for generating scanners.
- space[3]—an interpreter for an array definition language (ADL).
- grep[4]—a text search tool used to find one or more input files for lines containing a match to a specified pattern.
- sed[5]—a stream editor used to filter text.
- gzip[6]—a compression utility.
- make[7]—a tool for controlling the generation of executables.

Table 1 presents statistics on the number of predicates and their instances in the above programs. In Table 1, *#func* is the number of functions in a program. It can be seen that the number of predicates in each

[1] grep—http://www.gnu.org/software/grep/

[2] flex—http://flex.sourceforge.net
[3] space—http://sir.unl.edu/portal/bios/space.php
[4] grep—http://www.gnu.org/software/grep/
[5] sed—http://www.gnu.org/software/sed/
[6] gzip—http://www.gnu.org/software/gzip/
[7] make—http://sir.unl.edu/content/bios/make.php

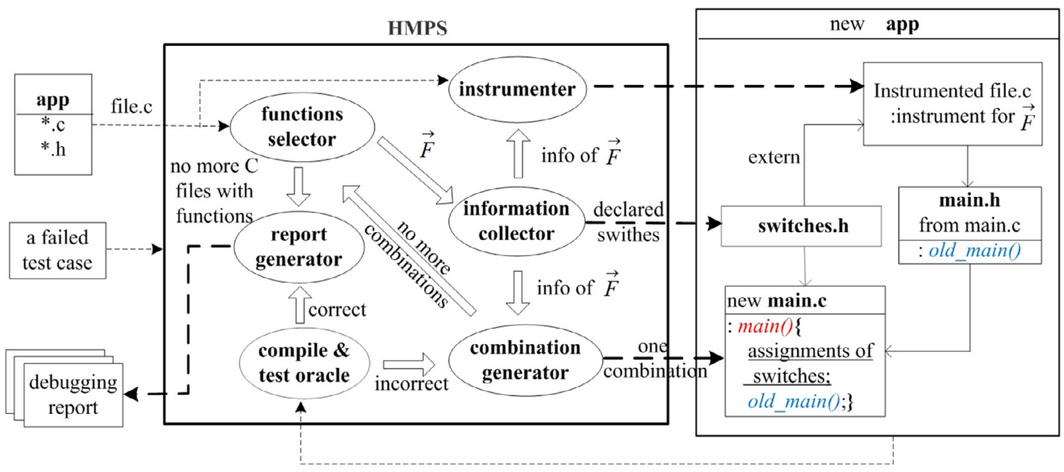

Fig. 1. Architecture of HMPS.

Table 1
Predicates and instances in programs.

Program	kLOC	#func	#pred	p_max	≤ 10	i_max	≤ 20
flex	11	162	816	82	93%	8010	77%
space	6	136	503	73	91%	958	84%
grep	10	146	955	95	90%	15,940	75%
sed	15	255	601	93	90%	3051	91%
gzip	6	104	724	75	94%	129,338	98%
make	36	268	1032	144	92%	5027	83%

program (#pred) ranges from 503 to 1032, while the maximum number of predicates in a function (p_max) ranges from 73 to 144. For each program, more than 90% of the functions contain at most 10 predicates (≤10), and the maximum number of predicate instances in a function in a randomly selected failed run (i_max) ranges from 958 to 129,338. However, the percentage of functions containing at most 20 predicate instances (≤20) is at least 75%, and it can be as large as 98%.

The statistics shows that it is infeasible to take all combinations of predicate instances in a run into consideration, because the minimum number of predicate instances in an execution (i_max) of the programs is 958, and hence there are 2^{958} different combinations of predicate instances. However, the percentage of functions containing at most 20 predicate instances (≤20) is at least 75%. Consequently, it confirms that a possible efficient way to implement multiple predicate switching is to search for the critical predicate sets in pieces of a program, like a function or two call-dependent functions.

3. HMPS

In this section, we first depict the architecture of our method. We then describe the algorithm together with several function selection strategies and switch combination strategies to accelerate the search for critical predicate sets. We last illustrate the instrumentation method and the complexity of HMPS.

3.1. The architecture of HMPS

As shown in Figure 1, the input of HMPS is the source code of a faulty application *app* and a failed test case, and the output of HMPS encompasses statements that include the critical predicates and statements on which the critical predicates are data or control dependent. In the following, we describe six key components of HMPS.

- *Function selector* selects one or several highly suspect functions, denoted by $\vec{F}$, from a C file *file.c* of *app* with DCC (Perez et al., 2014), a spectrum-based component ranking technique. If HMPS cannot identify any critical predicate in any single function, it generates combinations of functions among these highly suspect functions. Moreover, the combination process will proceed in a hierarchical fashion by following the call graph of *app*. If all highly suspect functions and their combinations are tried without success to find a CPS, HMPS proceeds to the report generator.
- *Information collector* collects predicate locations and the number of instances of each predicate in $\vec{F}$. For non-crash faults, HMPS employs a test code coverage tool to collect the above information of predicates, like GCOV,[8] a test code coverage tool being used in conjunction with GCC. As for crash faults, HMPS does static analysis of $\vec{F}$ to collect locations of predicates. After the collection, switch arrays of predicates can be constructed, declared and initialized in *switches.h*. This head file will be included by every C file containing any selected function. By doing this, an execution path of *app* can be changed when states of some predicate instances are flipped.
- *Instrumenter* instruments switches for predicates in $\vec{F}$ in line with the predicate information gathered by *Information collector*. Since switches need to be instrumented within statements, HMPS does instrumentation at the source code level. Moreover, this facilitates the exploration of switch combination strategies. The instrumentation method will be depicted in detail in Section 3.
- *Combination generator* generates a combination of switches of predicate instances in $\vec{F}$. The generated switch combination works as follows. Firstly, the original *main.c* is rewritten as *main.h* with *main()* function renamed as *old_main()*. The *switches.h* and *main.h* will be included by a newly formed *main.c* file, and the *old_main()* will be invoked by the function *main()*. Secondly, for the switches to take effect, elements of the switch array in the combination will be assigned the value 1 in the new *main()* function. If no more switch combinations are available, HMPS returns to the *function selector* to get a new function combination.
- *Compile* and *test oracle* recompiles the new *app* after each new *main.c* file being created, and tests it with the same input again. If the output is correct, then the CPS is identified and HMPS proceeds to the report generator. Otherwise, HMPS returns to the *combination generator*.
- *Report generator* generates a debugging report depending on whether the CPS is identified or not. If the CPS is captured, then statements containing the critical predicates are first reported to developers as the most suspect statements. Statements in $\vec{F}$ on which the CPS are data or control dependent are then returned as the second part of the localization result. In addition, all executed statements in functions with higher suspiciousness than functions in $\vec{F}$ will be reported as well. If a CPS is not identified, all executed

[8] GCOV—https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

statements in the failed run will be returned to developers. Furthermore, the returned statements are ordered by the ranking of their functions and the order they appear in their functions.

```
Algorithm 1: HMPS_Debugging.
input  : app: the source code of a faulty application
         FTC: a failed test case
output: a debugging report
1  crit_pred ← ∅
2  foreach file.c in app do
3      app_cov ← ∅
4      if the failure is not a crash then
5          func_sel ← DCC(file.c, FTC)
6          app_cov ← GCOV(app, FTC, file.c)
7      else
8          func_sel ← getFunCallDep(file.c, FTC)
9          app_cov ← SCOV(app, FTC, file.c)
10     func_combs ← enumByDepend(func_sel)
11     foreach $\vec{F}$ in func_combs do
12         (info_$\vec{F}$, switches.h)← collect_info(app_cov, $\vec{F}$)
13         new_file.c ← instrument(instr_strat, file.c, info_$\vec{F}$)
14         switches ← genSwitches($\vec{F}$)
15         switch at most m instances simultaneously
16         if crit_pred ≠ ∅ then
17             return genReport(crit_pred)
18 return failed
```

3.2. The algorithm of HMPS

Our algorithm, HMPS_Debugging, is as shown in Algorithm 1. It has two inputs: the source code of a faulty application *app* and a failed test case *FTC*. The algorithm uses *crit_pred* to store the CPIS (critical predicate instance set) and initializes it as the empty set (line 1). For each C file *file.c* in *app*, HMPS does the following.

First, *app_cov* (line 3) is initialized as the empty set. It is to store the location of each predicate in *file.c* and the number of instances of each predicate in the faulty run.

If the bug causes a non-crash fault, HMPS employs DCC, a spectrum-based component ranking technique, to find the highly suspect functions in the faulty program (line 5). The names of these highly suspect functions are stored in *func_sel*. In DCC, the following Ochiai is applied to rank functions in a program *P*:

$$S_O(f) = \frac{a_{11}(f)}{\sqrt{(a_{11}(f) + a_{01}(f)) \times (a_{11}(f) + a_{10}(f))}} \quad (1)$$

where $a_{ij}(f) = |\{e|x_{ef} = i \wedge r_e = j\}|$, $i, j \in \{0, 1\}$ and f is a function in *P*. Also, $x_{ef} = i$ indicates whether function f has been invoked ($i = 1$) in the execution of run e or not ($i = 0$). Similarly, $r_e = j$ represents whether a run e was faulty ($j = 1$) or not ($j = 0$). This similarity coefficient ranks functions of *P* with respect to their likelihood of containing bugs in *P*. In addition, the locations of predicates and their execution times in the failed execution are collected by a dynamic coverage collect tool, GCOV (line 6). Otherwise, if the bug causes a crash fault, obtain the call graph of *file.c* via getFunCallDep(*file.c*) (line 8), and gather the locations of predicates in *file.c* using static analysis.

Next, HMPS assigns to *func_combs* the sequence of function combinations from *func_sel* (line 10), where each combination consists of at most three functions to avoid a combination explosion. The function enumByDepend(*func_sel*) generates function combinations via the following three strategies:

1. Single function *F*. If the bug causes a non-crash fault, functions are ordered by the suspicious calculated by using formula (1). Otherwise, *F* will be selected according to a breadth-first search of the call graph.
2. Combinations of the form $\{F_1, F_2\}$ where both F_1 and F_2 are selected using Strategy 1, and F_1 calls F_2.
3. Combinations of the form $\{F_1, F_2, F_3\}$ where each of them is selected using Strategy 1, and one of the following holds: F_1 calls F_2 and F_2 calls F_3, and F_1 calls both F_2 and F_3.

For each function combination $\vec{F}$ in *func_combs*, HMPS does the following (lines 11–13). First, it reads the predicate information of $\vec{F}$ from *app_cov* and stores it in *info_*$\vec{F}$. According to *info_*$\vec{F}$, the head file *switches.h* is created to declare a linear switch array for each predicate. The size of each switch array is determined by the number of instances of the corresponding predicate. Then, *new_file.c* is formed by instrumenting *file.c*, according to the instrumentation strategy (Section 3.3) in *instr_strat*. Function genSwitches($\vec{F}$) uses the following strategies to generate switch combinations (line 14).

1. Loop predicates usually have more than one instance in a run. However, there is no need to simultaneously switch multiple *TRUE* instances of the same loop predicate, because the loop will be terminated after the state of the first instance being altered. In our experiments, we found that this can reduce a great number of useless switch combinations, which saves a lot of search time.
2. For conditional predicates, *i.e.*, *if* predicates, we apply the following two switch combination strategies.
   Combination strategy I: Simply assign 1 to all elements of a switch array, which means states of all instances of an *if* predicate will be altered simultaneously. In this way, HMPS may find the critical predicates for operator mutation faults effectively when the *if* predicate has many instances. Furthermore, this reduces greatly the size of search space, especially when the fault cannot be eliminated by altering states of the *if* predicate instances.
   Combination strategy II: Different instances of an *if* predicate are switched separately. When a conditional statement is in a loop structure or in a function which is called more than once in the faulty run, this predicate in the conditional statement may have multiple instances. Hence, more accurate combinations of predicate instances can be produced by this strategy.

Since the algorithm switches at most m ($m = 3$ in our experiments) instances simultaneously, HMPS terminates the debugging process in polynomial time. As stated in the *Report generator* part of Section 3.1, a debugging report will be generated according to the critical predicate set *crit_pred* (lines 16–17).

3.3. Instrumentation

To facilitate the implementation of switch combination strategies, HMPS needs to instrument within statements that contain selected predicates. Therefore, we develop an instrumenter at the source code level and present two instrumentation strategies. Besides, to avoid segmentation faults caused by flipped states of predicate instances, we propose an approach to simplify the instrumentation for loop predicates without any side effect.

Before doing the instrumentation, we should know which statements contain executed predicates in the faulty run and how many instances each executed predicate has. The answer of the former one determines where to do instrumentation and the latter one answers what will be instrumented. For non-crash faults, statements containing predicates and their execution times can be easily extracted from the coverage information of programs generated via using a test code coverage tool. If the bug causes a crash fault, HMPS does static analysis to collect locations of predicates and sets initial sizes of the switch arrays. If a CPS cannot be identified, the sizes of the switch arrays will be gradually adjusted. In our implementation, the size of the switch array for a branch (resp. loop) predicate is initialized as 5 (resp. 10), and is increased by 5 (resp. 10) each time when it is reset.

In the following, we show two instrumentation strategies for predicates and predicate instances.

3.3.1. *Predicate instance switches*

Suppose that there is a given program P, a failed test case FTC for P, and a function F of P. The instrumentation of F would proceed as follows. Run P on FTC; Let $e_0, e_1, \ldots, e_{n-1}$ be all the predicates in function F to be instrumented. For each predicate e_i, HMPS declares a global Boolean array $sw_i[m_i]$ and a global index variable num_i for sw_i in the head file *switches.h*. Here, m_i is the execution times of e_i plus 1 or 2 (to deal with off-by-1 or off-by-2 bugs related to the number of the execution times of a loop body), and each element of sw_i and num_i are initialized as 0. As a result, each e_i is replaced with

$$sw_i[num_i++]?\,!e_i : e_i$$

in function F, where $!e_i$ is the negation of e_i. Note that num_i is increased by 1 when the statement is executed. Therefore, the state of the jth instance of e_i is switched when the value of $sw_i[j]$ is 1.

This also applies to predicates in nested loop structures. For example, the following piece of code is a function for the selection sort, where there are two faults at line 5 and 7.

```
1: void sort(int a[], int N) {
2:   for(i=0; i<N; i++){
3:     k=i;
4:     for(j=i+1; j<N ;j++)
       // correct version: a[j]<a[k]
5:       if(a[j]>a[k])
6:         k=j;
7:     if(k<i){ // correct version: k!=i
8:       temp=a[i]; a[i]=a[k]; a[k]=temp;
9:     }}}
```

Assume that all predicates in the function are executed in a failed run, then the functions is instrumented as follows:

```
1: void sort(int a[], int N) {
2:   for(i=0; sw1[num1++]?!(i<N):i<N; i++){
3:     k=i;
4:     for(j=i+1; sw2[num2++]?!(j<N):j<N;j++)
5:       if(sw3[num3++]?!(a[j]>a[k]):a[j]>a[k])
6:         k=j;
7:     if(sw4[num4++]?!(k<i):k<i){
8:       temp=a[i]; a[i]=a[k]; a[k]=temp;
9:     }}}
```

Although line 4 (5 or 7) appears in a nested loop, the predicate at that line can also be easily instrumented using a linear switch array as defined above.

3.3.2. *Predicate switches*

However, the above instrumentation strategy may result in useless combinations of predicate instances in some cases, especially when the predicate has many instances and all the instances need to be switched simultaneously. For example, the following piece of code describes a fault at line 2543 in sed-4.1.5.

```
 (p)  (F)  (LN)
  -:   -:  2539: static inline void
116: 112:  2540: output_missing_newline(outf)
  -:   -:  2541:   struct output *outf;
  -:   -:  2542: {
                   //correct version: if(outf->missing_newline)
116: 112:  2543:    if (!outf->missing_newline)
  -:   -:  2544:    {
  0: 112:  2545:      ck_fwrite("\n", 1, 1, outf->fp);
  0: 112:  2546:      outf->missing_newline = false;
  -:   -:  2547:    }
116: 112:  2548: }
```

In the faulty run, the predicate at line 2543 takes the *TRUE* branch in its all 112 executions, while it takes the *FALSE* branch in all its 116 instances in the correct run. Although the above instrumentation strategy lets the size of the switch array be one more than the number of predicate instances in the failed run, it cannot handle such cases where the number of predicate instances in the failed run is much smaller than that in the correct run. Moreover, HMPS will not succeed even if all combinations of predicate instances at line 2543 are considered. Consequently, another strategy is proposed to instrument switches for predicates, not for their instances. In the above case, if we instrument the predicate at line 2543 as follows:

```
if (sw?outf->missing_newline:!outf->missing_newline),
```

then the modified faulty run will produce correct results when `sw` is assigned 1.

Specifically, HMPS declares a global Boolean array sw_i for each predicate e_i in the head file *switches.h* and sw_i is initialized as 0. In function F, each e_i will be replaced with $sw_i ? \neg e_i : e_i$, in which $\neg e_i$ is the negation of e_i. Note that states of all instances of e_i are flipped when the value of sw_i is assigned 1. In our experiments, if a CPS cannot be found under the instrumentation of predicates, predicate instance switches will be instrumented to continue the search for a CPS.

3.3.3. *Simplification for loop predicates*

In our study, we found that predicate switching may lead to segmentation faults especially for loop predicates. This is caused by null pointers and array overruns. For instance, a fault in space[9] at line 6510 is shown as follows:

```
// correct version: while (app_ptr != NULL) {
6510: while (app_ptr->NEXT != NULL) {
                        ...
          app_ptr = app_ptr->NEXT;
}}
```

A correct execution of the program may be obtained by switching the last instance of the loop predicate at line 6510, but this may cause a segmentation fault, since *app_ptr* will be assigned null when the state of the last instance of the predicate is altered. To address this problem, we replace the predicate e_i with $sw_i[num_i++]?0:1$, as shown in the following instrumented code:

```
while (sw1[num1++]?0:1) {
    ...
    app_ptr = app_ptr->NEXT;
}
```

Why does it work? Suppose that a loop predicate e_i is executed t times, *i.e.*, the loop is executed $t-1$ times, which means that e_i is evaluated as *TRUE* in the first $t-1$ times and *FALSE* in the last time. If the assignment of $sw_i[t]$ is 1 and that of $sw_i[j]$ is 0 where $0 \le j \le t-1$, then the loop will be executed exactly t times in such cases. In this way, the segmentation fault caused by the predicate at line 6510 can be avoided. Therefore, if the loop predicate does not involve any assignment, 0 (1) can be used to replace $!e_i$ (e_i) directly. However, this way of simplification is only suitable for loop predicates that do not have any side effect.

[9] space—http://sir.unl.edu/portal/bios/space.php

Table 2
Experimental subjects.

Subject	Version	kLOC	Test cases	Coverage
flex	2.5.33	12	525	74.6%
grep	2.6	10.2	669	59.73%
gzip	1.3.13	5.7	215	64.17%
sed	4.1.5	14.5	370	64.21%
space	ORACOLO2	6.2	174	92.08%

Table 3
Fault injectors used.

Name	Function
Negate decision (ND)	Negate the condition in an if or loop statement
Replace constant (RC)	Off-by-one replacement of integer constant
Delete statement (DS)	Delete a statement
Replace operator (RO)	Replace an operator by another operator
Replace variable (RV)	Replace a variable in a statement
Multiple fault (MF)	A composition of above injectors

Table 4
Fault injectors for generating errors in predicates.

Name	Function (in conditions)
Negate decision (PND)	Negate the condition in an if or loop statement
Replace variable (PRV)	Replace a variable in predicates
Replace operator (PRO)	Replace an operator in predicates
Replace constant (PRC)	Replace a constant in predicates
Delete branch (PDB)	Delete subbranches for predicates
Add branch (PAB)	Add subbranches for predicates

3.4. Complexity of HMPS

Multiple predicate switching based fault localization is intractable when taking all combinations of predicate instances into consideration. For HMPS, since at most m predicate instances in each function combination $\vec{F}$ are switched simultaneously, the time complexity of Algorithm 1 is $O(kC_n^m)$, in which n is the number of predicate instances in $\vec{F}$, k is the number of considered functions and functions combinations, and m can be adjusted in practice ($m = 3$ in our experiments).

The space complexity of HMPS is $O(N \cdot M)$ because it is determined by the number of involved predicates (N) and the length of each switch linear array (M), *i.e.* the number of instances of each such predicate.

4. Experimental investigation

In this section, a set of empirical studies are conducted to evaluate the performance of HMPS for real programs. Firstly, we present the experimental setup. Secondly, we discuss our experimental results, and two examples are presented to show that how critical predicates can offer useful clues for finally pinpointing the root cause. We finish this section with a discussion of threats to validity.

4.1. Experimental setup

In this part, we present subjects, evaluation metrics and comparison methods that are used in our experiments.

4.1.1. Subjects

In our experiments, five subjects written in C program language were recruited:

- flex[10]—a tool for generating scanners.
- grep[11]—a text search tool used to find one or more input files for lines containing a match to a specified pattern.
- gzip[12]—a compression utility.
- sed[13]—a stream editor used to filter text.
- space[14]—an interpreter for an array definition language (ADL).

Table 2 presents the details of each subject. The LOC count information was gathered by the tool SLOCCount.[15] Test cases and coverage percentage were collected with GCOV.

To evaluate the performance of HMPS in localizing both single faults and multiple faults, our experiments were performed using 60 faulty versions per program. Since these programs are bug-free, we injected common mistakes in the programs using various fault injectors shown in Table 3 and Table 4. Each fault injector (Steimann et al., 2013) in Table 3 was used to create five injections into each subject. In our experiments, we found that HMPS could capture CPSes for all 25 decision negated (ND) faults out of 150 faulty versions generated by injectors in Table 3. Therefore, to assess the performance of HMPS on bugs occurring in conditions of conditional and loop statements, we injected predicate bugs in the subjects by injectors in Table 4. Also, each fault injector in Table 4 was used to create five faulty versions for each subject. In total, we evaluated 300 faulty versions.

A passed and a failed test suite for each program are randomly generated, and statement-based reduction strategy (Yu et al., 2008) is used to reduce the size of each test suite. In Table 2, the number of used test cases and the coverage of each subject are respectively listed in the column of *Test cases* and *Coverage*.

4.1.2. Metrics

The metrics gathered were the percentage of the number of LOCs needed to be inspected until the developer hits the faulty code. HMPS first reports statements that encompass critical predicates. It then reports statements on which critical predicates are data or control dependent and in functions with higher suspiciousness than functions containing critical predicates. If HMPS could not find any critical predicate for a faulty version, then all executed statements in a faulty run will be reported as the localization result. All captured statements will be ranked according to the suspiciousness of their own functions and the locations that they appeared in the functions. This metric assumes that developers inspect localization results in above ordered manner, starting from the top ranking statements.

4.1.3. Comparison methods

We compared our method with Barinel and Ochiai, because Barinel tactfully combines spectrum-based fault localization techniques with model-based diagnosis methods and Ochiai is one of the state-of-the-art statistical fault localization approaches. In our experiments, we subject both Barinel and Ochiai to above described 300 common bugs of programs in Table 2 and evaluate their localization accuracy using the above mentioned metrics.

4.2. Results and discussion

In the following, we first depict the results of our method on 300 common bugs by comparing with both Barinel and Ochiai. We then separately show the results on common types of bugs and bugs occurred in conditional and loop statements.

4.2.1. Overall results of HMPS

Figure 2 divides these 300 common bugs into three categories: Zero, One and Multiple. In which, “Zero” is the number of bugs that HMPS cannot find any critical predicate for them, “One” represents the

[10] flex—http://flex.sourceforge.net
[11] grep—http://www.gnu.org/software/grep/
[12] gzip—http://www.gnu.org/software/gzip/
[13] sed—http://www.gnu.org/software/sed/
[14] space—http://sir.unl.edu/portal/bios/space.php
[15] sloccount—http://www.dwheeler.com/sloccount/

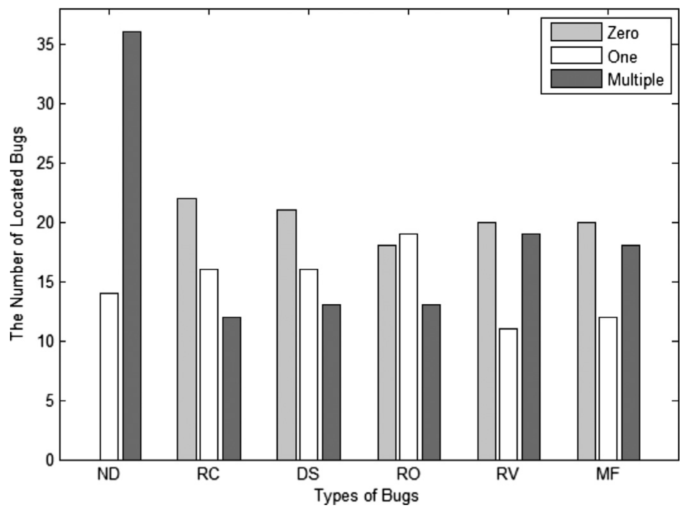

Fig. 2. The number of bugs located by HMPS.

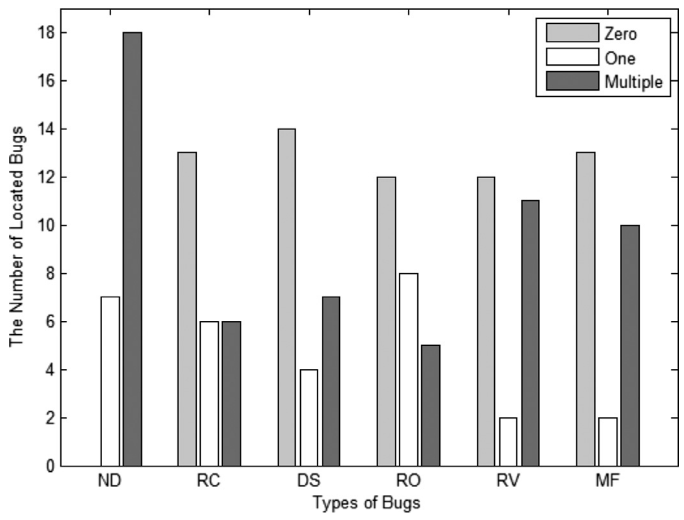

Fig. 4. The number of different types of bugs located by HMPS.

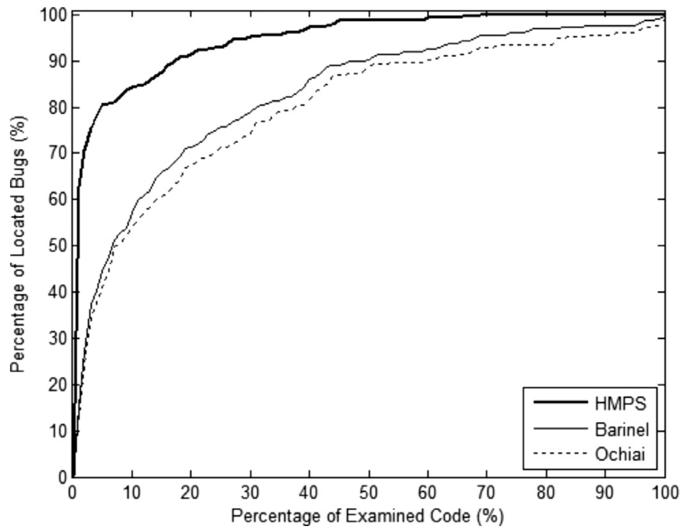

Fig. 3. Results of HMPS on 300 common bugs.

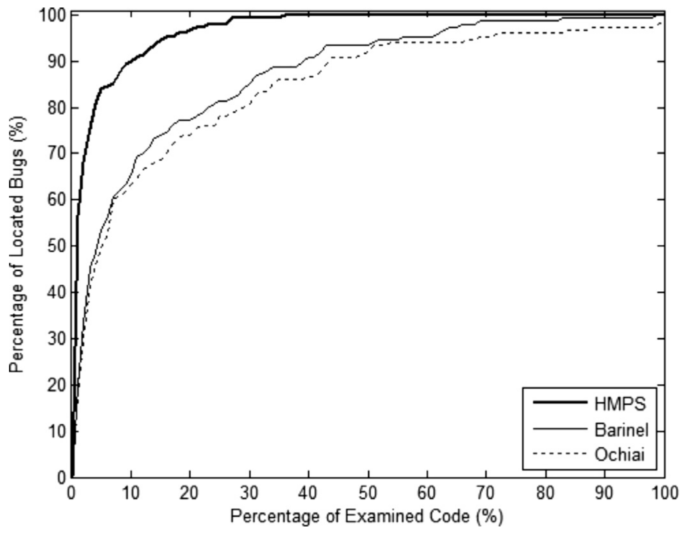

Fig. 5. Results of HMPS on 150 common bugs.

number of bugs that the CPS can be found by HMPS through switching a single predicate instance, and “Multiple” means the number of bugs that the CPS only can be captured by HMPS through switching multiple predicate instances. As we can see from Figure 2, the number of “Multiple” bugs is more than that of “One” bugs for ND, RV and MF faults (mentioned in Table 3). Accordingly, to extend the application of predicate switching, it is necessary to explore multiple predicate switching. In total, HMPS finds critical predicates for 199 out of 300 bugs, in which 111 out of the 199 bugs are captured by switching multiple predicate instances.

Figure 3 compares the results of HMPS with both Barinel and Ochiai on these 300 common bugs. The x-axis of each plot represents the percentage of statements in the faulty program to be examined, and the y-axis is the percentage of bugs located within the given code examination range. For instance, (10, 85) is that 85% faults can be located by HMPS when 10% of the code is examined. As we can see from Figure 3, HMPS consistently outperforms (*i.e.*, more faults for the same percentage of the examined code) Barinel and Ochiai over all the examined range. Besides, HMPS can identify 81% of 300 common bugs in five real-life utility programs when examining up to 4% of the code, while Barinel and Ochiai locate 44% and 39% of these 300 bugs respectively.

4.2.2. Results on various types of common bugs

Similarly, Figure 4 illustrates how HMPS identify CPSes for six types of 150 common bugs (mentioned in Table 3), in which the number of “Multiple” bugs is more than that of “One” bugs for ND, DS, RV and MF faults. HMPS catches the CPS for 86 out of 150 such bugs, in which 57 out of 86 bugs solved by simultaneously switching more than one predicate instance.

Figure 5 depicts the overall results of HMPS on these 150 bugs. It can be seen that HMPS consistently achieves better average fault localization results than Barinel and Ochiai in the entire examined range. In detail, Figure 5 tells us that for these 150 bugs, when a developer would like to examine at most 5% of the code, HMPS catches 85% of the bugs while Barinel and Ochiai locate 55% and 51% respectively. Furthermore, when 27% code examination is acceptable, Barinel and Ochiai respectively identify about 82% and 77% out of the 150 bugs, while our method locates 99% of the bugs.

Figure 6 shows the results of comparing HMPS with Barinel and Ochiai on bugs of six different types. We observe that HMPS consistently achieves better results for these six types of bugs injected by injectors in Table 3. Also, HMPS captures all bugs of any specific type by examining less than 37% of the code. Besides, HMPS noticeably outperforms Barinel and Ochiai for ND, DS and MF bugs. For ND bugs, the failed run might be easily changed to the correct execution through altering states of negated predicates, no matter how many

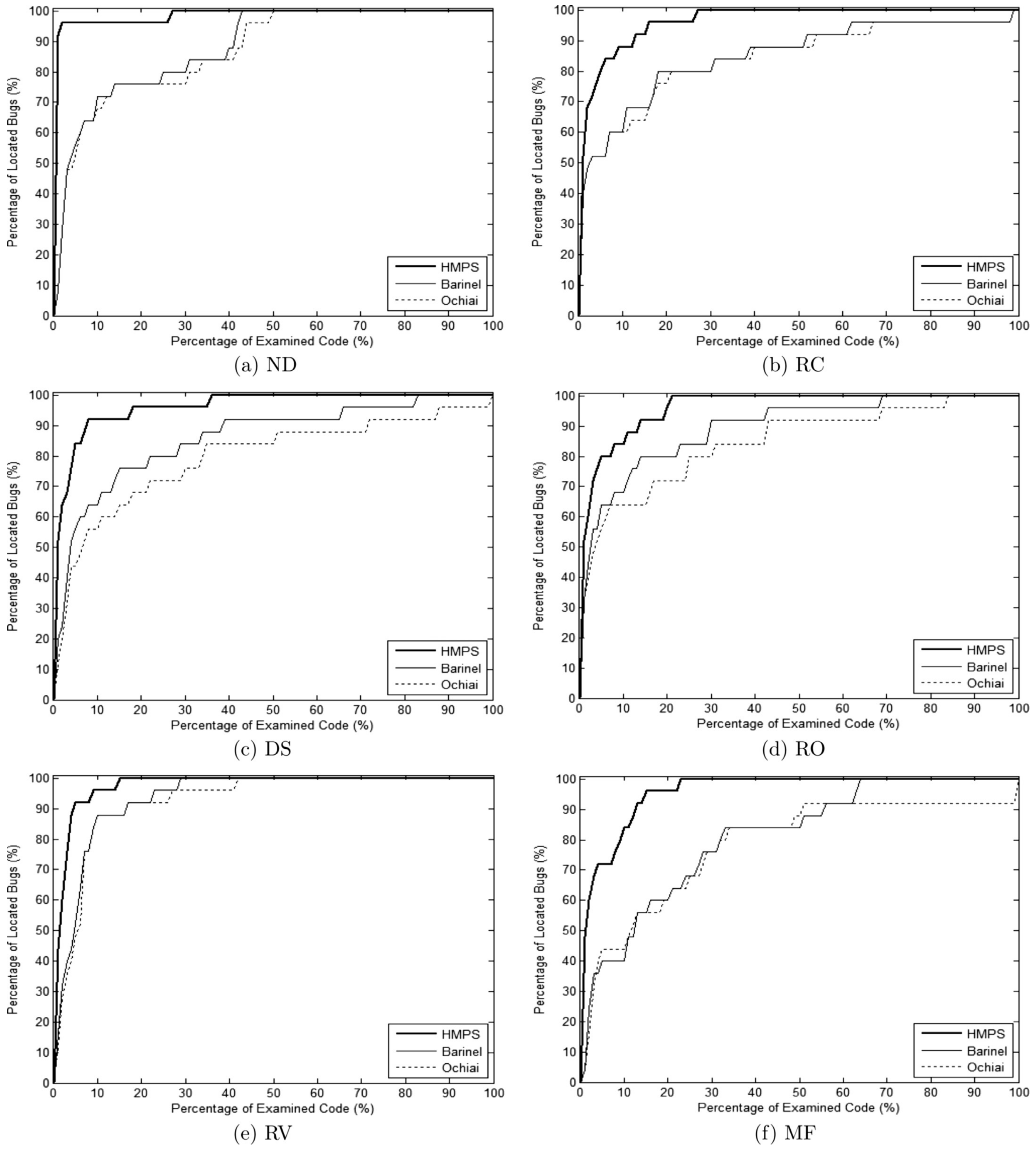

Fig. 6. Comparison of HMPS, Barinel and Ochiai on different types of bugs from five subjects.

instances the predicate should have. For DS bugs in Figure 6(c), although HMPS cannot find any critical predicate for 14 out of 25 such bugs (see Figure 4), we find that the compared result is even better than that of ND bugs, because of the highly ranking of the functions containing the faulty code in most cases. In addition, when 8% code examination is acceptable, HMPS locates more than 90% of the 25 DS bugs, while Barinel and Ochiai identify 64% and 56% respectively. As for MF bugs in Figure 6(f), although HMPS failed to find any critical predicate for some multiple faults, HMPS can also obtain better finally localization results for them. As stated for DS bugs, the reason is that functions containing the faulty code are highly suspect for these faulty versions. For the other MF bugs, HMPS captures CPSes for them, because the CPS can be captured for each single bug of the compound bugs. Moreover, all 25 multiple bugs can be located by HMPS and it shows great advantage over Barinel and Ochiai when examining up to 2% or 15% of the code.

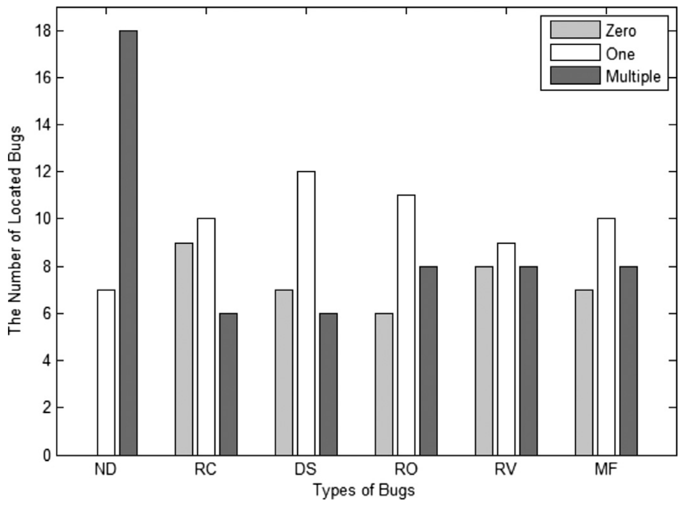

Fig. 7. The number of different types of predicate bugs located by HMPS.

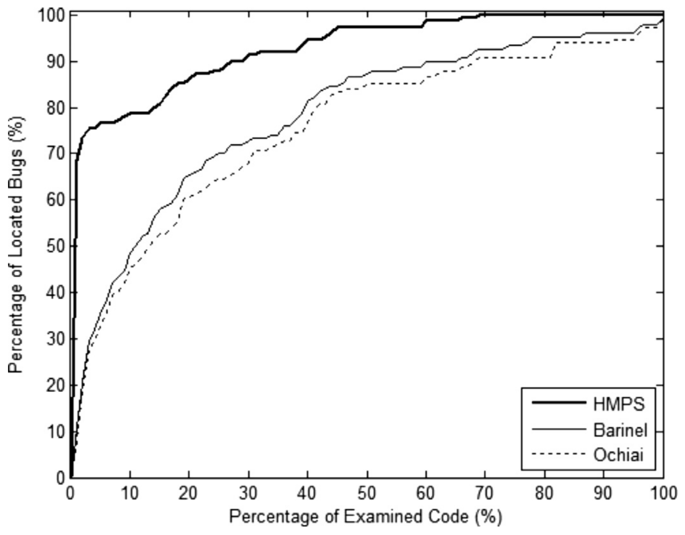

Fig. 8. Results of HMPS on 150 bugs in predicates.

4.2.3. *Results on predicate bugs*

In Figure 4, we can see that HMPS can identify the CPS for all ND bugs that occurred in conditions of conditional or loop statements. As we know, there are various types of common faults occurred in conditions. Accordingly, we extended our experiments by increasing anther 150 common predicate bugs generated by injectors in Table 4.

Also, Figure 7 illustrates the number of bugs that how their CPSes can be identified by HMPS. As we can see, the number of “Zero” bugs is reduced greatly among the predicate bugs, because HMPS can capture CPSes for 113 out of 150 predicate bugs. In addition, the number of “One” bugs and the number of “Multiple” bugs are almost the same, which also confirms that predicate switching techniques in its applications need to be extended by exploring multiple predicate switching.

Figure 8 depicts the overall results of HMPS on these 150 predicate bugs. We can see that HMPS consistently achieves better average locating results than Barinel and Ochiai in the entire examined range. For instance, when examining 3% of the code, HMPS can find nearly 75% of the bugs, while Barinel and Ochiai identify about 29% and 27% of the bugs respectively. As we can see from Figure 8, nearly 92% of the bugs can be located by HMPS when examining up to 30% of the code.

In Figure 9, we compare the outcomes of HMPS with Barinel and Ochiai on predicate bugs of six different types. As we can see, HMPS outperforms these two methods on the six types of predicate bugs. Moreover, HMPS takes great advantage over Barinel and Ochiai on PNN, PRO, PRC, PDB and PAB bugs. For PNN bugs in Figure 9(a), HMPS locates more than 95% such bugs via only examining 2% of the code, while Barinel and Ochiai capture around 35% of such bugs. For PRC bugs in Figure 9(d), when a user examine 2% of the code, HMPS can identify nearly 75% of the bugs, while Barinel and Ochiai both capture around 17% of such bugs. For PDB bugs in Figure 9(e), the PDB bug may cause the number of instances of the predicate reduced in disjunction expressions and increased in conjunction expressions. In our experiments, we found that HMPS can identify CPSes for bugs that the number of predicate instances is increased or reduced by one, because currently HMPS only increases the size of any switch array by one. As we can see in Figure 9(e), HMPS can identify nearly 60% of the bugs when only examining up to 2% of the code, while Barinel and Ochiai both capture around 13% of such bugs. For PAB bugs, they may cause the number of instances of the buggy predicate increased when adding a branch through a disjunction and reduced when adding a branch through a conjunction. The reason is the same as above stated for PDB bugs. In Figure 9(f), 68% of the PAB bugs can be captured by HMPS when the examined code reaches up to about 3% of the code, while Barinel and Ochiai respectively identify 30% and 27% of such bugs.

4.3. *Explanatory capabilities*

HMPS alters the states of multiple predicate instances to bring the program execution to a successful completion. Hence, the switched predicates may be good inspection start points for further locating the root cause, and the statements that are data or control dependent with these switched predicates may offer useful clues for finally pinpointing the root cause. In our experiments, we find that we can identify the root cause via following above reported statements. We illustrate this with two bugs from our experiments.

The following piece of code describes a bug in space,[16] where there is an omission fault at line 2453 of file *space.c*. It causes some unexpected outputs of the variable THEA_PHEPOL in several failed runs. In our experiments, HMPS finds a CPIS with two elements: one is the second instance of the predicate at line 2332, and the other is the only instance of the predicate at line 2351.

```
2321: int fixport(struct Elem *elem_ptr)
2323: { ...
2329:   app_ptr = elem_ptr->PORT_PTR;
2332:   while(app_ptr != NULL){
        ...
2350:   if(elem_ptr->POLARIZATION == LIN_POL)
2351:       if(app_ptr->OMIT_POL == YES)
2352:           ...
2368:   app_ptr=app_ptr->NEXT;
      }}

2377: int fixselem(struct Elem *elem_ptr)
2379: { ...
2430:   if(elem_ptr->NPORTS == 0){
2433:       port_ptr=(struct Port*)malloc(sizeof(struct Port));
            ...
2453:       //port_ptr->OMIT_POL = YES; (omission fault)
2455:       port_ptr->NEXT = NULL;
2457:       elem_ptr->NPORTS = 1;
2458:       elem_ptr->PORT_PTR = port_ptr;
2459:   };
2462:   fixport(elem_ptr);
```

The tracking process is started from one of the switched instance, *i.e.* the predicate at line 2332. We trace the instance to the first

[16] space—http://sir.unl.edu/portal/bios/space.php

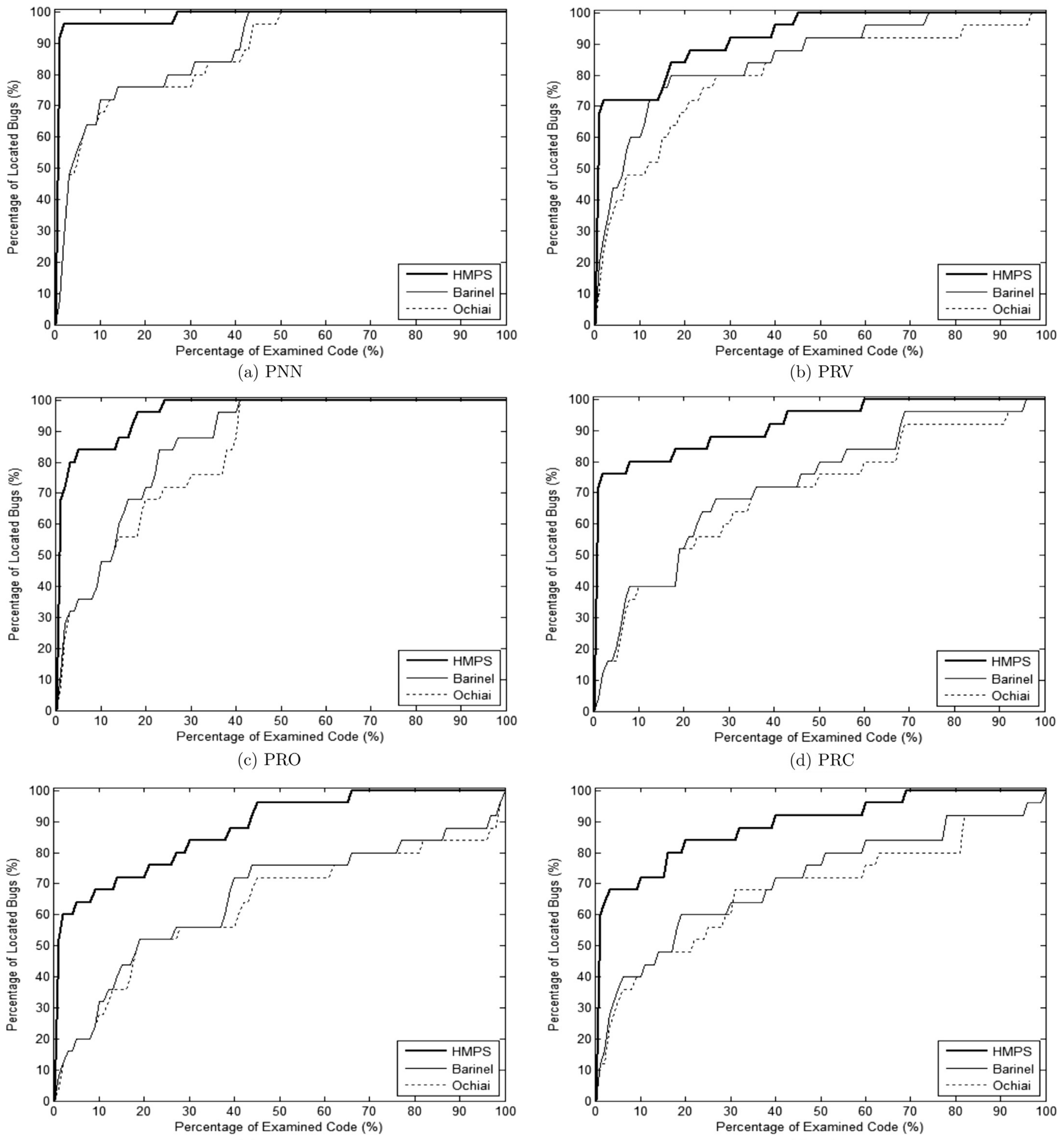

Fig. 9. Comparison of HMPS, Barinel and Ochiai on different types of bugs in predicates from five subjects.

execution of line 2368, and watch `app_ptr→NEXT`. Then, we trace line 2368 back to line 2329 which assigns `elem_ptr→ PORT_PTR` to `app_ptr`, and now watch `(elem_ptr→PORT_PTR) → NEXT`. Note that `elem_ptr` is the only parameter of function `fixport()`, which is called by function `fixselem()` at line 2462. Therefore, this brings us to line 2458, and our watch point is shifted to `port_ptr →NEXT`, which makes us suspicious of line 2455 that causes the bug.

For another switched instance, since it is the only instance of predicate at line 2351, we trace this instance to line 2329 and watch `(elem_ptr→ PORT_PTR) → OMIT_POL`. By using the same reasoning as the above, we are brought to line 2462. This brings us to look at line 2458, and our watch point is shifted to `port_ptr→OMIT_POL`, which makes us look for an assignment of `port_ptr→OMIT_POL`. However, no assignment statement about `port_ptr →OMIT_POL` is in function `fixselem()`, hence there are decent reasons to suspect that an omission assignment occurred in this function, which is exactly the root cause of the bug.

From this example, we can see that the latter switched predicate can provide useful clue for developers to find the root cause and the former one can make us find statements that are closely related to the root cause without much effort.

The second example describes a bug in sed-4.1.5,[17] where line 1111 and 1112 in the following code should be exchanged. It fails to produce an output file for some inputs. In our experiments, HMPS finds a CPIS with five predicate instances: the first instances of the predicates at lines 1118, 1135 and 1137, the 14th instance of the predicate at line 1235, and the 7th instance of the predicate at line 2933. Here we just explain how users trace from the predicate at line 2933 with little understanding of the original program. Since the variable *c* is one of the parameters of function `add1_buffer()` that is called by function `compile_filename()` at line 1112, then variable *ch* in function `compile_filename()` is now observed. This makes us suspicious of the assignment of *ch* at line 1111, which is exactly the root cause of the bug. It again shows that one of the switched predicate instances may provide useful clues for developers to find the root cause.

```
1093: static void compile_filename(...){
        ...
1105:   while(ch!=EOF && ch!='\n'){
1111:       ch = inchar();     //correct: add1_buffer(b,ch);
1112:       add1_buffer(b,ch); //correct: ch = inchar();
        } ...
1118:   for(p=file_ptrs; p; p=p->link)
1119:       if(p->readit_p==readit &&
               strcmp(p->name,file_name)==0)
1120:           break;
1135:   if(name) ...
1137:   if(fp) ...
      }
1231: static int inchar(){
1233:   int ch = EOF;
1235:   if(prog.cur) {...}
1240:   else if(prog.file) {...} ...
1251:   return ch;
      }}
2923: void add1_buffer(struct buffer *b, int c){
2933:   if(c != EOF){...}
      }
```

4.4. Threats to validity

The main threat to external validity of the empirical results is the fact that only injected faults are considered in our evaluations. Although 60 common bugs in total for each subject were generated by injectors in Table 3 and Table 4, these faulty versions may not represent all types of bugs in reality.

The main threat to internal validity lies in the randomly selected failed test cases. Since HMPS instruments switch arrays according to the execution times of predicates in a failed run, thus the randomly selected failed test case may not offer proper coverage information for combinations. For instance, the following piece of code presents an operator mutation bug at line 868 in sed-4.1.5,[18]

```
(P)  (F)  (LN)  //correct: while((ch=inchar())!=EOF && ch!='\n')
20:   2:  868:  while ((ch = inchar()) == EOF && ch != '\n')
 -:   -:  869:  {
16:   0:  870:     pending_mb = BRLEN (ch, &cur_stat) != 1;
                   ...
16:   0:  873:     if (!pending_mb)
 -:   -:  874:         {...}
                 }
```

[17] sed—http://www.gnu.org/software/sed/

[18] sed—http://www.gnu.org/software/sed/

In the selected faulty run, we find that the predicate takes the *FALSE* branch two times, while it takes the *TRUE* branch in 16 out of its 20 instances in the same test of the correct version. HMPS cannot find any critical predicate for the bug with the coverage information, because statements in the while loop body need to be executed 16 times in the run of the correct version, but the predicate at line 868 only has two instances. HMPS cannot handle such cases by instrumentation methods mentioned in Section 3.3.2. We may benefit from trying other failed test cases. However, this may be a very time-consuming process, because there may not exist a coincidental path in the program. In addition, if HMPS cannot find any critical predicate for a suite of tests, then one solution is to try other failed tests which generates different code coverage. Therefore, in our future work, it would be better to explore some test case selection strategies to alleviate the threat to such cases.

Another threat to internal validity is that HMPS cannot handle global variable initialization faults because it does multiple predicate switching in functions instead of the whole program.

5. Related work

There are a great deal of related works of software debugging in recent decades. In this section, we only concentrate on several closely related works, like statistics-based approaches, state-altering methods, model-based approaches, and spectrum-based reasoning methods.

Statistics-based fault localization techniques use program spectra, proposed by Reps et al. (1997), to find a statistical relationship with observed failures. Many different forms of blocks whose execution results comprise the program spectra, like statements, branches, data dependencies, executive paths (Chilimbi et al., 2009), predicate and components of programs. Jones et al. (2002) proposed the technique that rank executed statements in a faulty program according to their occurrence in failed and passed runs of the program and developed a tool, called Tarantula, to visualize the localization results, which can assist the developer in quickly identifying root causes. Later, Santelices et al. (2009) compared different coverage of programs, like statements, branches and data dependencies, and found that the cost of combinations of coverage is less than using every single coverage type. Similarly, Jaccard (Chen et al., 2002) and Ochiai (Abreu et al., 2006) are two statistical methods that can be applied in block hit spectrum-based fault localization tools. Abreu et al. (2007) showed that Ochiai can provide more accurate locating results than Tarantula and Jaccard. The nearest neighbor technique (Renieres and Reiss, 2003) finds differences between the program statement spectra of one failed execution and one passed execution. This method is effective especially when it is applied along with nearest neighbor queries. DCC (Perez et al., 2014) uses progressively detailed program spectra, from components to statements, to locate faults with less execution overhand. The Cooperative Bug Isolation (CBI; Liblit et al., 2003, 2005) and Sober (Liu et al., 2005) collect and pass evaluations of predicates and function return values on to a statistical engine to find predicates that are highly correlated with failures. HOLMES (Chilimbi et al., 2009) uses path profiles with more rich information than predicate profiles. Actually, paths are an instance of compound predicates (Arumuga Nainar et al., 2007). Our method first applies DCC to calculate the suspiciousness of each function in a faulty program.

State-alerting approaches (Gore and Reynolds, 2012) attempt to find root causes of faulty programs according to changing states of variables in an execution or comparing states of variables in two distinct but similar runs. Delta debugging (Cleve and Zeller, 2005) finds a minimal set of program states, which when changed, can change a pass execution into a fail one. It then finds the cause transitions by simplification (Zeller and Hildebrandt, 2002) and isolation (Choi and Zeller, 2002) algorithms. Simplification obtains the minimum set of circumstances that lead to the failure. Isolation simplifies a difference between a successful configuration and a failed configuration. Zeller (2002) isolated the cause-effect chain of a failure via comparing memory graphs captured in distinct executions of the faulty program, and identified cause transitions as the likely locations of the defect. Single predicate switching (Zhang et al., 2006) changes the state of one predicate instance every time to find a path that can outputs the correct results. Instead of altering execution paths, Jeffrey et al. (2008) proposed a value replacement technique that alters values of a set of instances of program variables using collected alternate values. To outputs correct results, our method flips the states of compound predicates in functions of a failed execution.

Model-based software debugging (MBSD) is an application of the model-based diagnosis (MBD) technique to debug computer programs with bugs (Mayer and Stumptner, 2007). The techniques use program-independent search algorithm on a logical model of the programs to find the minimal set of statements whose incorrectness can explain incorrect outcomes when the program is executed on the given test case. Model-based software debugging was first presented by Wotawa et al. (2002). This approach automatically extracts a logical model from the program without further user interaction and without a formal specification. However, this technique cannot scale to non-trivial utility programs, thus MBSD techniques are used to refine the output yielded by spectrum-based fault localization techniques (Mayer et al., 2008; Abreu et al., 2009a). Abreu et al. (2009b) proposed spectrum-based reasoning technique, named Barinel, via tactfully combining spectrum-based fault localization with model-based diagnosis. Abreu et al. (2009b) constructed the model of a program using the coverage information of the program, and applies Bayesian reasoning to deduce the diagnostic candidates. In addition, model checking techniques can be used to search for the minimal set of statements (Ball et al., 2003; Griesmayer et al., 2007; Groce, 2004; Groce et al., 2004; Liu and Li, 2010). For example, Groce (2004) proposed to call the model checker CBMC (Clarke et al., 2004) to get a failing run, and find the differences between the failed run and a passed run generated by using a pseudo-Boolean solver. Later, Liu and Li (2010) proposed the bounded multiple predicate switching (BMPS) technique by resorting to CBMC, a bounded model checker for C programs, to find critical predicates for complex faults in small programs. It locates faults through switching the outcomes of instances of multiple predicates to get a successful execution where each loop is executed for a bounded number of times. However, CBMC does not scale to non-trivial utility programs. To extend the application of predicate switching, our method does not do any abstraction of the program. Instead, our method changes a failed run to a passing one via switching states of multiple predicate instances in the source code and checks the outputs by testing the modified code directly. Also, instrumentation and combination strategies are proposed in this paper to facilitate the search for critical predicates.

6. Conclusions

In this paper, we have presented a hierarchical multiple predicate switching method, named HMPS, that can locate complex faults in utility programs via identifying critical predicates. The fundamental idea is to restrict the search for critical predicates to highly suspect functions generated by SBFL techniques and following the call dependency graph. To explore switch combination strategies readily, a source code instrumenter is developed to instrument linear switch arrays for involved predicates. Instrumentation and switch combination strategies are proposed to facilitate the search for critical predicate sets (CPSes). Our empirical studies show that HMPS can identify CPSes for 199 out of 300 common bugs in five real-life utility programs, while single predicate switching only works for 88 out of the 300 bugs.

We now emphasize three advantages of HMPS inherited from single predicate switching. First, only one failed test case is needed in the stage of identifying critical predicates. This advantage gives us the additional benefit that information about this failed test case such as values of variables and execution times of statements can be provided to developers to enhance the explanatory ability of the localization results. Second, critical predicates are good inspecting start points for further analysis. Third, HMPS can offer useful clues for finally pinpointing the faults because statements that are data or control dependent with critical predicates are parts of the localization results.

Our future work will include finding more effective switch combination strategies and test case selection strategies to make HMPS more efficient and effective. Another future direction is to improve our work in the aspect of the explanatory capability of localization results.

Acknowledgment

This work is supported by State Key Laboratory of Software Development Environment Open Fund (SKLSDE-2012KF-08).

References

Abreu, R., Mayer, W., Stumptner, M., van Gemund, A.J.C., 2009a. Refining spectrum-based fault localization rankings. In: Proceedings of the ACM Symposium on Applied Computing (SAC'09), pp. 409–414.
Abreu, R., Zoeteweij, P., Van Gemund, A.J., 2006. An evaluation of similarity coefficients for software fault localization. In: Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06), pp. 39–46.
Abreu, R., Zoeteweij, P., Van Gemund, A.J., 2007. On the accuracy of spectrum-based fault localization. In: Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, pp. 89–98.
Abreu, R., Zoeteweij, P., Van Gemund, A.J., 2009b. Spectrum-based multiple fault localization. In: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering, ASE 2009, pp. 88–99.
Arumuga Nainar, P., Chen, T., Rosin, J., Liblit, B., 2007. Statistical debugging using compound boolean predicates. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'07), pp. 5–15.
Ball, T., Naik, M., Rajamani, S.K., 2003. From symptom to cause: Localizing errors in counterexample traces. In: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03), pp. 97–105.
Chen, M.Y., Kiciman, E., Fratkin, E., Fox, A., Brewer, E., 2002. Pinpoint: Problem determination in large, dynamic internet services. In: Proceedings of International Conference on Dependable Systems and Networks (DSN'02), pp. 595–604.
Chilimbi, T.M., Liblit, B., Mehra, K., Nori, A., Vaswani, K., 2009. Holmes: Effective statistical debugging via efficient path profiling. In: Proceedings of 31st IEEE International Conference on Software Engineering (ICSE'09), pp. 34–44.
Choi, J., Zeller, A., 2002. Isolating failure-inducing thread schedules. In: ACM SIGSOFT Software Engineering Notes (SEN'02), pp. 210–220.
Clarke, E., Kroening, D., Lerda, F., 2004. A tool for checking ansi-c programs. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04), pp. 168–176.
Cleve, H., Zeller, A., 2005. Locating causes of program failures. In: Proceedings of the 27th International Conference on Software engineering (ICSE'05), pp. 342–351.
Gore, R., Reynolds, P.F., Jr 2012. Reducing confounding bias in predicate-level statistical debugging metrics. In: Proceedings of the International Conference on Software Engineering (ICSE'12), pp. 463–473.
Griesmayer, A., Staber, S., Bloem, R., 2007. Automated fault localization for c programs. In: Electronic Notes in Theoretical Computer Science (ENTCS), pp. 95–111.
Groce, A., 2004. Error explanation with distance metrics. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04), pp. 108–122.
Groce, A., Kroening, D., Lerda, F., 2004. Understanding counterexamples with explain. In: Proceedings of International Conference on Computer Aided Verification (CAV'04), pp. 318–321.
Jeffrey, D., Gupta, N., Gupta, R., 2008. Fault localization using value replacement. In: Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA'08), pp. 167–178.
Jones, J.A., Harrold, M.J., Stasko, J., 2002. Visualization of test information to assist fault localization. In: Proceedings of the 24th International Conference on Software Engineering (ICSE'02), pp. 467–477.
Liblit, B., Aiken, A., Zheng, A.X., Jordan, M.I., 2003. Bug isolation via remote program sampling. In: ACM SIGPLAN Notices, pp. 141–154.
Liblit, B., Naik, M., Zheng, A.X., Aiken, A., Jordan, M.I., 2005. Scalable statistical bug isolation. In: ACM SIGPLAN Notices, pp. 15–26.
Liu, C., Yan, X., Fei, L., Han, J., Midkiff, S.P., 2005. Sober: statistical model-based bug localization. In: ACM SIGSOFT Software Engineering Notes (SEN'05), pp. 286–295.
Liu, Y., Li, B., 2010. Automated program debugging via multiple predicate switching. In: Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI'10), pp. 327–332.

Mayer, W., Abreu, R., Stumptner, M., van Gemund, A.J.C., 2008. Prioritizing model-based debugging diagnostic reports. In: Proceedings of the 19th International Workshop on Principles of Diagnosis (DX'08), pp. 127–134.

Mayer, W., Stumptner, M., 2007. Model-based debugging c state of the art and future challenges. In: Electronical Notes in Theoretical Computer Science (ENTCS'07), pp. 61–82.

Parnin, C., Orso, A., 2011. Are automated debugging techniques actually helping programmers? In: Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA'11), pp. 199–209.

Perez, A., Abreu, R., Riboira, A., 2014. A dynamic code coverage approach to maximize fault localization efficiency. Journal of Systems and Software, 90, 18–28.

Renieres, M., Reiss, S.P., 2003. Fault localization with nearest neighbor queries. In: Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03), pp. 30–39.

Reps, T., Ball, T., Das, M., Larus, J., 1997. The use of program profiling for software maintenance with applications to the year 2000 problem. In: Proceedings of the 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'99), pp. 432–449.

Santelices, R., Jones, J.A., Yu, Y., Harrold, M.J., 2009. Lightweight fault-localization using multiple coverage types. In: Proceedings of the 31st International Conference on Software Engineering (ICSE'09), pp. 56–66.

Steimann, F., Frenkel, M., Abreu, R., 2013. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In: Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA'13), pp. 314–324.

Wotawa, F., Stumptner, M., Mayer, W., 2002. Model-based debugging or how to diagnose programs automatically. Proceedings of the 15th International Conference on Industrial and Engineering, Application of Artificial Intelligence and Expert Systems (IEA/AIE'02), pp. 746–757.

Yu, Y., Jones, J.A., Harrold, M.J., 2008. An empirical study of the effects of test-suite reduction on fault localization. In: Proceedings of the 30th International Conference on Software engineering (ICSE'08), pp. 201–210.

Zeller, A., 2002. Isolating cause-effect chains from computer programs. In: Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE'02), pp. 1–10.

Zeller, A., Hildebrandt, R., 2002. Simplifying and isolating failure-inducing input. IEEE Transactions on Software Engineering 28 (2), 183–200.

Zhang, X., Gupta, N., Gupta, R., 2006. Locating faults through automated predicate switching. In: Proceedings of the 28th International Conference on Software Engineering (ICSE'06), pp. 272–281.

Xiaoyan Wang received her M.E. degree from Sun Yat-sen University, China in 2010. She is currently a Ph.D. candidate at Sun Yat-sen University of School of Information Science Technology. Her research interests include software engineering issues on program analysis, testing, debugging and web service composition.

Yongmei Liu is a professor of Computer Science at Sun Yat-sen University, China. She received her Ph.D. degree in Computer Science from University of Toronto in 2006. Her research interests lie in artificial intelligence, knowledge representation and reasoning, cognitive robotics, program verification and debugging.