Scalable Distributed Concolic Testing: A Case Study on a Flash Storage Platform*

Yunho Kim, Moonzoo Kim, and Nam Dang

CS Dept. KAIST
Daejeon, South Korea
{kimyunho,moonzoo}@kaist.ac.kr,
paddy@kaist.ac.kr

Abstract. Flash memory has become a virtually indispensable component for mobile devices in today's information society. However, conventional testing methods often fail to detect hidden bugs in flash file systems due to the difficulties involved in creating effective test cases. In contrast, the approach of model checking guarantees a complete analysis, but only on a limited scale. In the previous work, the authors applied concolic testing to the multi-sector read operation of a Samsung flash storage platform as a trade-off between the aforementioned two methods.

This paper describes our continuing efforts to develop an effective and efficient verification framework for flash file systems. We developed a scalable distributed concolic algorithm that utilizes a large number of computing nodes. This new concolic algorithm can alleviate the limitations of the concolic approach caused by heavy computational cost. We applied the distributed concolic technique to the multi-sector read operation of a Samsung flash storage platform and compared the empirical results with results obtained with the original concolic algorithm.

1 Introduction

On the strengths of characteristics such as low power consumption and strong resistance to physical shock, flash memory has become a crucial component for mobile devices. Accordingly, in order for mobile devices to operate successfully, it is imperative that the flash storage platform software (e.g., file system, flash translation layer, and low-level device driver) operates correctly. However, conventional testing methods often fail to detect hidden bugs in flash storage platform software, since it is difficult to create effective test cases (i.e., test cases that provide a check of all possible execution scenarios generated from complex flash storage platform software). Thus, the current industrial practice of manual testing does not achieve high reliability or provide cost-effective testing. As another testing approach, randomized testing can save human effort for test case generation. However, it does not achieve high reliability, because random input

* This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST)/National Research Foundation of Korea(NRF) (Grant 2010-0001727) and the MKE(Ministry of Knowledge Economy), Korea, under the ITRC(Information Technology Research Center) support program supervised by NIPA(National IT Industry Promotion Agency) (NIPA-2010-(C1090-1031-0001)).

A. Cavalcanti et al. (Eds.): ICTAC 2010, LNCS 6255, pp. 199–213, 2010.
© Springer-Verlag Berlin Heidelberg 2010

data does not necessarily guarantee high coverage of a target program. These deficiencies of conventional testing incur significant overhead to manufacturers. In spite of the importance of flash memory, however, little research work has been conducted to formally analyze flash storage platforms. In addition, most of such work [10,7] has focused on the specifications of file system design, not real implementation.

In the previous work [12], the authors applied *concolic* (CONCrete + symbOLIC) testing [16,8,4] (also known as dynamic symbolic execution [18] or automated white-box fuzzing [9]) to the multi-sector read operation (MSR) of the Samsung OneNAND flash storage platform [15] and tested all possible execution paths in an *automatic and exhaustive* manner. We used CREST [3] (an open source concolic testing tool for C) in the experiments and confirmed that concolic testing was effective to detect bugs. However, CREST consumed a large amount of time to analyze all possible execution paths, which is not acceptable in an industrial setting. For example, it took more than three hours to test a MSR with a small explicit environment consisting of 5 physical units and 6 logical sectors, which generated 2.8×10^6 test cases in total. Although concolic testing effectively detects bugs through the full path coverage, the required heavy computational cost prohibits the use of concolic testing in real world applications.

This paper describes our continuing efforts to develop an effective and efficient verification framework for flash file systems by alleviating the limitations caused by heavy computational cost. One solution is to develop a *scalable distributed concolic algorithm* that can utilize a large number of computing nodes with high efficiency. Thus far, most of automated formal verification techniques such as model checking have suffered heavy computational costs. Consequently, this heavy overhead often prevents practitioners from adopting these valuable techniques. The concolic approach is a suitable technique to exploit the benefits of parallel computing. We modified the original concolic algorithm to utilize multiple computing nodes in a distributed manner so as to reduce time cost significantly. In addition, this distributed concolic algorithm is scalable to utilize a large number of computing nodes, achieving linear speedup with an increasing number of computing nodes. We applied this distributed concolic technique on the multi-sector read operation (MSR) of a Samsung flash storage platform with 16 computing nodes. This paper reports experimental results obtained with the new concolic approach and compares them with the results derived with the original concolic algorithm to demonstrate the former's performance gain and scalability.

The organization of this paper is as follows. Section 2 explains the original concolic testing algorithm. Section 3 describes the distributed concolic algorithm. Section 4 overviews the multisector-read (MSR) function of the Samsung flash storage platform. Section 5 presents the experimental results obtained by applying the distributed concolic algorithm to MSR. Section 6 concludes the paper along with directions for future work.

2 Original Concolic Testing Algorithm

This section presents the original concolic testing algorithm [16,8,4]. Concolic testing executes a target program both concretely and symbolically [14,19] at the same time. Concolic testing proceeds via the following five steps:

1. *Instrumentation*
   A target C program is statically instrumented with probes, which record symbolic path conditions (PCs) from a concrete execution path when the target program is executed. Note that PCs correspond to conditional statements (i.e., `if`) in the target program.
2. *Concrete execution*
   The instrumented C program is executed with given input values and the concrete execution part of the concolic execution constitutes the normal execution of the program. For the first execution of the target program, the initial inputs are assigned with random values. For the second execution and onward, input values are obtained from step 5.
3. *Obtaining a symbolic path formula ϕ_i*
   The symbolic execution part of the concolic execution collects symbolic path conditions over the symbolic input values at each branch point encountered along the concrete execution path. Whenever each statement s of the target program is executed, a corresponding probe inserted at s updates the symbolic map of symbolic variables if s is an assignment statement, or collects a corresponding symbolic path condition pc, if s is a branch statement. Thus, a complete symbolic path formula ϕ_i of the ith execution is the conjunction of all PCs $pc_1, pc_2, ...pc_n$ where pc_j is executed earlier than pc_{j+1} for all $1 \leq j < n$.
4. *Generating a symbolic path formula ϕ'_i for the next input values*
   Given a symbolic path formula ϕ_i obtained in Step 3, to obtain the next input values, ϕ'_i is generated by negating the path condition pc_j (initially $j = n$) and removing

Input:
$path$: a sequence of PCs executed in the previous execution
neg_limit: a position of a PC in $path$ beyond which PCs should not be negated
Output:
a set of generated test cases (i.e., I's of line 7)

```
1  Concolic(path, neg_limit) {
2  j =| path | ;
3  while j >= neg_limit do
4  |   // φ is a symbolic path formula of path
5  |   // pc_k is kth path condition of path and pc_1 is executed first
6  |   φ = pc_1 ∧ ... ∧ pc_{j-1} ∧ ¬pc_j ;
7  |   I = SMT_Solver(φ) // returns NULL if φ is unsatisfiable
8  |   if I is not NULL then
9  |   |   path' = execute a target program on I ;
10 |   |   Concolic(path', j + 1);
11 |   end
12 |   j = j − 1;
13 end
14 }
```

Algorithm 1. Original concolic algorithm

the subsequent PC (i.e., $pc_{j+1}, ...pc_n$) of ϕ_i. If ϕ'_i is unsatisfiable, another path condition pc_{j-1} is negated and the subsequent PCs are removed, until a satisfiable path formula is found. If there are no further available new paths, the algorithm terminates.

5. *Selecting the next input values*
   A constraint solver such as a Satisfiability Modulo Theory (SMT) solver [17] generates a model that satisfies ϕ'_i. This model decides concrete next input values and the entire concolic testing procedure iterates from Step 2 again with these input values.

Algorithm 1 describes the original concolic algorithm in detail, which corresponds to Step 2 to Step 5. Algorithm 1 negates all PCs of a given path one by one in descending order (see line 3 to line 13) and new paths ($path'$ in line 9) are analyzed recursively (see line 10). To prevent redundant analysis of a given path, subsequent recursive $Concolic()$ negates PCs of $path'$ up to neg_limit th PC (i.e., only $pc_{|path'|}, pc_{|path'|-1}, ..., pc_{|neg_limit|}$ of $path'$ are negated one by one). Note that this concolic algorithm operates in a similar manner to the depth first order (DFS) traversal of the execution tree of a target program.

3 Distributed Concolic Algorithm

This section describes a *distributed* concolic algorithm that can utilize a large number of computing nodes. The main concept underlying the new algorithm is based on the feature that symbolic path formulas in the loop (line 3 to line 13 of Algorithm 1) of the original concolic algorithm are analyzed *independently*. Therefore, in order to analyze these symbolic path formulas in a distributed manner, Algorithm 2 generates and stores symbolic path formulas in $queue_{pf}$ (line 15) without analyzing these symbolic path formulas recursively (line 10 of Algorithm 1). If $queue_{pf}$ is empty (exiting the loop of line 5 to line 25) and there are no more paths to analyze in all distributed nodes, the algorithm terminates (line 31). Otherwise, the current node requests a symbolic path formula from another node n' (line 27) and receives a symbolic path formula from n' (line 28). The received symbolic path formula is then added into $queue_{pf}$ (line 29) and the algorithm continues from line 5 again. If the current node receives a request for symbolic path formulas (line 17), it sends one from $queue_{pf}$ (lines 19 and 20) immediately as long as $queue_{pf}$ is not empty.[1]

Note that communication between nodes occurs only when $queue_{pf}$ is empty. Since $queue_{pf}$ is non-empty for most of the analysis time, the number of communications is small compared to the number of analyzed symbolic path formulas. In addition, the communicated message contains only one symbolic path formula, whose size is small (proportional to the length of the corresponding execution path). Furthermore, this algorithm is not affected by the complexity and/or characteristics of a target program. Therefore, Algorithm 2 is scalable to utilize a large number of computing nodes without performance degradation.

[1] In a real implementation, there is a server to coordinate communications between computing nodes; this is not described in this paper for the sake of providing a simple description.

```
Input:
orig_path: a sequence of PCs executed in the previous execution
Output:
a set of generated test cases (i.e., I's of line 12)
1  DstrConcolic(orig_path) {
2  queue_pf = ∅; // queue containing symbolic path formulas
3  Add (orig_path, 1) to queue_pf;
4  repeat
5      while queue_pf is not empty do
6          Remove (path, neg_limit) from queue_pf;
7          j =| path |;
8          while j >= neg_limit do
9              // φ is a symbolic path formula of path
10             // pc_k is kth path condition of path and pc_1 is executed first
11             φ = pc_1 ∧ ... ∧ pc_{j-1} ∧ ¬pc_j ;
12             I = SMT_Solver(φ); // returns NULL if φ is unsatisfiable
13             if I is not NULL then
14                 path' = execute the target program on I;
15                 Add (path', j + 1) to queue_pf;
16             end
17             if there is a request for a symbolic path formula from other node n then
18                 if queue_pf is not empty then
19                     Remove (path'', neg_limit'') from queue_pf;
20                     Send (path'', neg_limit'') to n;
21                 end
22             end
23             j = j − 1;
24         end
25     end
26     if there are uncovered paths in any distributed node then
27         Send a request for a symbolic path formula to n' whose queue_pf is not empty;
28         Receive (path, neg_limit) from n';
29         Add (path, neg_limit) to queue_pf;
30     end
31 until all execution paths are covered;
32 }
```

Algorithm 2. Distributed concolic algorithm

4 Overview of Multi-sector Read Operation

Unified storage platform (USP) is a software solution to operate a Samsung flash memory device [15]. USP allows applications to store and retrieve data on flash memory through a file system. USP contains a flash translation layer (FTL) through which data and programs in the flash memory device are accessed. The FTL consists of three layers - a sector translation layer (STL), a block management layer (BML), and a low-level

device driver layer (LLD). Generic I/O requests from applications are fulfilled through the file system, STL, BML, and LLD, in order. MSR resides in STL. [2]

4.1 Overview of Sector Translation Layer (STL)

A NAND flash device consists of a set of *pages*, which are grouped into *blocks*. A *unit* can be equal to a block or multiple blocks. Each page contains a set of *sectors*.

When new data is written to flash memory, rather than overwriting old data directly, the data is written on empty physical sectors and the physical sectors that contain the old data are marked as invalid. Since the empty physical sectors may reside in separate physical units, one logical unit (LU) containing data is mapped to a linked list of physical units (PU). STL manages this mapping from logical sectors (LS) to physical sectors (PS). This mapping information is stored in a sector allocation map (SAM), which returns the corresponding PS offset from a given LS offset. Each PU has its own SAM.

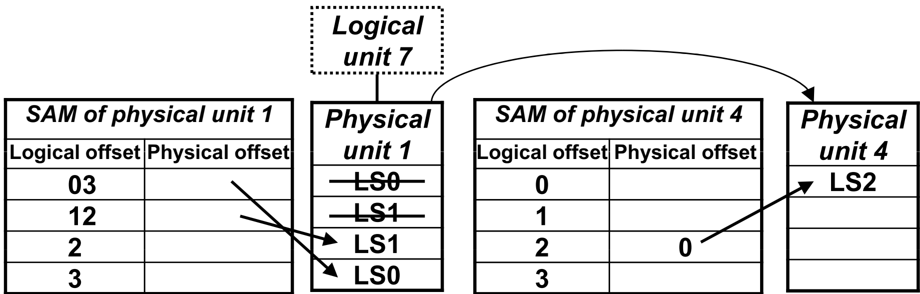


Fig. 1. Mapping from logical sectors to physical sectors

Figure 1 illustrates a mapping from logical sectors to physical sectors where 1 unit consists of 1 block and 1 block contains 4 pages, each of which consists of 1 sector. Suppose that a user writes LS0 of LU7. An empty physical unit PU1 is then assigned to LU7, and LS0 is written into PS0 of PU1 (SAM1[0]=0). The user continues to write LS1 of LU7, and LS1 is subsequently stored into PS1 of PU1 (SAM1[1]=1). The user then updates LS1 and LS0 in order, which results in SAM1[1]=2 and SAM1[0]=3. Finally, the user adds LS2 of LU7, which adds a new physical unit PU4 to LU7 and yields SAM4[2]=0.

4.2 Multi-sector Read Operation

USP provides a mechanism to simultaneously read as many multiple sectors as possible in order to improve the reading speed. The core logic of this mechanism is implemented in a single function in STL. Due to the non-trivial traversal of data structures for logical-to-physical sector mapping (see Section 4.1), the function for MSR is 157 lines long and

[2] This section is taken from [11].

highly complex, having 4-level nested loops. Figure 2 describes simplified pseudo code of these 4-level nested loops. The outermost loop iterates over LUs of data (line 2-18) until the `numScts` amount of the logical sectors are read completely. The second outermost loop iterates until the LSes of the current LU are completely read (line 5-16). The third loop iterates over PUs mapped to the current LU (line 7-15). The innermost loop identifies consecutive PSes that contain consecutive LSes in the current PU (line 8-11). This loop calculates `conScts` and `offset`, which indicate the number of such consecutive PSes and the starting offset of these PSes, respectively. Once `conScts` and `offset` are obtained, `BML_READ` rapidly reads these consecutive PSes as a whole (line 12).

```
01:curLU = LU0;
02:while(numScts > 0) {
03:  readScts = # of sectors to read in the current LU
04:  numScts -= readScts;
05:  while(readScts > 0 ) {
06:    curPU = LU->firstPU;
07:    while(curPU != NULL ) {
08:      while(...) {
09:        conScts=# of the consecutive PSes to read in curPU
10:        offse =the starting offset of the consecutive PSes
11:      }
12:      BML_READ(curPU, offset, conScts);
13:      readScts = readScts - conScts;
14:      curPU = curPU->next;
15:    }
16:  }
17:  curLU = curLU->next;
18:}
```

Fig. 2. Loop structures of MSR

For example, suppose that the data is “ABCDEF” and each unit consists of four sectors and PU0, PU1, and PU2 are mapped to LU0 (“ABCD”) in order and PU3 and PU4 are mapped to LU1 (“EF”) in order, as depicted in Figure 3(a). Initially, MSR accesses SAM0 to find which PS of PU0 contains LS0(‘A’). It then finds SAM0[0]=1 and reads PS1 of PU0. Since SAM0[1] is empty (i.e., PU0 does not have LS1(‘B’)), MSR moves to the next PU, which is PU1. For PU1, MSR accesses SAM1 and finds that LS1(‘B’) and LS2(‘C’) are stored in PS1 and PS2 of PU1 consecutively. Thus, MSR reads PS1 and PS2 of PU1 altogether through `BML_READ` and continues its reading operation.

The requirement property for MSR is that the content of the read buffer should be equal to the original data in the flash memory when MSR finishes reading, as given by `assert(`$\forall i$`.LS[`i`]==buf[`i`])` inserted at the end of MSR.[3]

[3] [13] describes a systematic method to identify this test oracle for MSR.

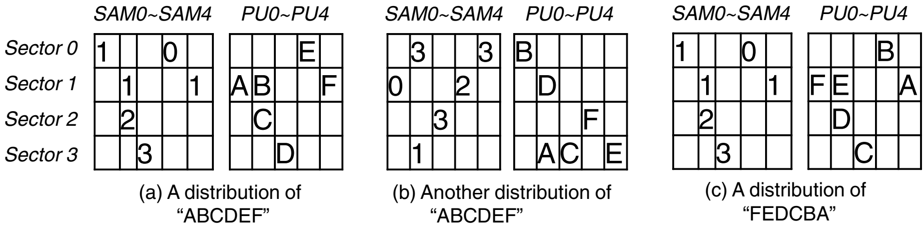


Fig. 3. Possible distributions of data "ABCDEF" and "FEDCBA" to physical sectors

In these analysis tasks, we assume that each sector is 1 byte long and each unit has four sectors. Also, we assume that data is a fixed string of distinct characters (e.g., "ABCDE" if we assume that data is 5 sectors long, and "ABCDEF" if we assume that data is 6 sectors long). We apply this data abstraction, since the values of logical sectors should not affect the reading operations of MSR, but the distribution of logical sectors into physical sectors does. For example, for the same data "ABCDEF", the reading operations of MSR are different for Figure 3(a) and Figure 3(b), since they have different SAM configurations (i.e., different distributions of "ABCDEF"). However, for "FEDCBA" in Figure 3(c), which has the same SAM configuration as the data shown in Figure 3(a), MSR operates in exactly same manner as for Figure 3(a). Thus, if MSR reads "ABCDEF" in Figure 3(a) correctly, MSR reads "FEDCBA" in Figure 3(c) correctly too.

In addition, we assume that data occupies 2 logical units. The number of possible distribution cases for l LSes and n physical units, where $5 \leq l \leq 8$ and $n \geq 2$, increases exponentially in terms of both n and l, and can be obtained by

$$\sum_{i=1}^{n-1} \left({}_{(4\times i)}C_4 \times 4! \right) \times \left({}_{(4\times(n-i))}C_{(l-4)} \times (l-4)! \right)$$

For example, if a flash has 1000 physical units with data occupying 6 LSes, there exist a total of 3.9×10^{22} different distributions of the data. Table 1 shows the total number of possible cases for 5 to 8 logical sectors and various numbers of physical units, respectively, according to the above formula.

Table 1. Total number of the distribution cases

PUs	4	5	6	7	8
$l = 5$	61248	290304	9.8×10^5	2.7×10^6	6.4×10^6
$l = 6$	239808	1416960	5.8×10^6	1.9×10^7	5.1×10^7
$l = 7$	8.8×10^5	7.3×10^6	3.9×10^7	1.5×10^8	5.0×10^8
$l = 8$	3.4×10^6	4.2×10^7	2.9×10^8	1.4×10^9	5.6×10^9

MSR has the characteristics of a control-oriented program (4-level nested loops) and a data-oriented program (large data structure consisting of SAMs and PUs) at the same time, although the values of PSes are not explicitly manipulated. As seen from Figure 3, the execution paths of MSR depend on the values of SAMs and the order of PUs linked to LU. In other words, MSR operates deterministically, once the configuration of the SAMs and PUs is fixed.

5 Case Study on Paralleized Concolic Testing of the Flash Storage Platform

In this section, we describe a series of experiments for testing the multisector read (MSR) operation of the unified storage platform (USP) for a Samsung OneNAND flash memory [15]. Also, we compare the empirical results of applying distributed concolic testing with the results of the original concolic testing [12].

Our goal is to investigate the distributed concolic algorithm, focusing on its performance improvement and scalability when applied to MSR. We thus pose the following research questions.

– **RQ1**: How does the distributed concolic algorithm improve the speed of concolic testing the MSR code?
– **RQ2**: How does the distributed concolic algorithm achieve scalability when applied to the MSR code?

5.1 Environment Model

MSR assumes that logical data are randomly written on PUs and the corresponding SAMs record the actual location of each LS. The writing is, however, subject to several constraint rules; the following are some of the representative rules. The last two rules can be enforced by the constraints in Figure 4.

1. One PU is mapped to at most one LU.
2. If the i_{th} LS is written in the k_{th} sector of the j_{th} PU, then the $(i \; mod \; m)_{th}$ offset of the j_{th} SAM is valid and indicates the PS number k, where m is the number of sectors per unit (4 in our experiments).
3. The PS number of the i_{th} LS must be written in *only* one of the $(i \; mod \; m)_{th}$ offsets of the SAM tables for the PUs mapped to the $\lfloor \frac{i}{m} \rfloor_{th}$ LU.

To enforce such constraints on test cases, a test driver/environment model generates valid (i.e., satisfying the environment constraints) test cases *explicitly* by selecting a PU and its sector to contain the l th logical sector (`PU[i].sect[j]=LS[l]`) and setting the corresponding SAM accordingly (`SAM[i].offset[l]=j`).

For example, Figure 3(a) represents the following distribution case:

– LS[0]=‘A’, LS[1]=‘B’, LS[2]=‘C’, LS[3]=‘D’, LS[4]=‘E’, and LS[5]=‘F’.
– PU[0].sect[1]=‘A’, PU[1].sect[1]=‘B’, PU[1].sect[2]=‘C’, PU[2].sect[3]=‘D’, PU[3].sect[0]=‘E’, and PU[4].sect[1] = ‘F’.

$$\forall i, j, k\ (LS[i] = PU[j].sect[k] \rightarrow (SAM[j].valid[i\ mod\ m] = true \\ \&\ SAM[j].offset[i\ mod\ m] = k \\ \&\ \forall p.(SAM[p].valid[i\ mod\ m] = false) \\ \text{where } p \neq j \text{ and } PU[p] \text{ is mapped to} \lfloor \frac{i}{m} \rfloor_{th}\ LU))$$

Fig. 4. Environment constraints for MSR

- SAM[0].valid[0]=true, SAM[1].valid[1]=true, SAM[1].valid[2]=true, SAM[2].valid[3]=true, SAM[3].valid[0]=true, and SAM[4].valid[1]=true (all other validity flags of the SAMs are false).
- SAM[0].offset[0]=1, SAM[1].offset[1]=1, SAM[1].offset[2]=2, SAM[2].offset[3] =3, SAM[3].offset[0]=0, and SAM[4].offset[1]=1.

Thus, the environment contraints for $i = 2, j = 1$, and $k = 2$ are satisfied as follows:

$$LS[2] = PU[1].sect[2] \rightarrow (SAM[1].valid[2\ mod\ 4] = true \\ \&\ SAM[1].offset[2\ mod\ 4] = 2 \\ \&\ SAM[0].valid[2\ mod\ 4] = false \\ \&\ SAM[2].valid[2\ mod\ 4] = false \\ \&\ SAM[3].valid[2\ mod\ 4] = false)$$

5.2 Test Setup for the Experiment

All experiments were performed on 64 bit Fedora Linux 9 equipped with a 3.6 GHz Intel Core2Duo processor and 16 gigabytes of memory. We utilized 16 computing nodes connected with a gigabit ethernet switch. We implemented Algorithm 2 in the open source concolic testing tool CREST [2]. However, since the CREST project is in its early stage, CREST has several limitations such as lack of support for dereferencing of pointers and array index variables in the symbolic analysis. Consequently, the target MSR code was modified to use an array representation of the SAMs and PUs. We used CREST 0.1.1 (with DFS search option), gcc 4.3.0, Yices 1.0.24 [5], which is an SMT solver used as an internal constraint solver by CREST for solving symbolic path formulas. Although CREST does not correctly test programs with non-linear arithmetic, we could apply CREST to MSR successfully, because MSR contains only linear integer arithmetic.

To evaluate the effectiveness of parallelized concolic testing (i.e., bug detecting capability), we applied *mutation analysis* [1] by injecting the following three types of frequently occuring bugs (i.e. mutation operators), as we did in our previous study [12]. The injected bugs are as follows:

1. *Off-by-1 bugs*
   - b_{11}: while(numScts>**0**) of the outermost loop (line 2 of Figure 2) to while(numScts>**1**)
   - b_{12}: while(readScts>**0**) of the second outermost loop (line 5 of Figure 2) to while(readScts>**1**)
   - b_{13}: for(i=0;i<conScts; i++) of BML_READ() (line 12 of Figure 2) to for(i=0;i<conScts**-1**;i++)
2. *Invalid condition bugs*
   - b_{21}: if(SAM[i].offset[j]**!=**0xFF) in the third outermost loop to if(SAM[i].offset[j]**==**0xFF)
   - b_{22}: readScts=((4-j)>numScts)?numScts:4-j in the innermost loop to readScts=((4-j)<numScts)?numScts:4-j
   - b_{23}: if((firstOffset+nScts)**==**SAM[i].offset[j]) in the innermost loop to if((firstOffset+nScts)**!=**SAM[i].offset[j])
3. *Missing statement bugs*
   - b_{31}: missing nScts=1 in the second outermost loop
   - b_{32}: missing nReadScts-- in the second outermost loop
   - b_{33}: missing nLun++ corresponding the line 17 of Figure 2

To evaluate the efficiency of parallelized concolic testing, we measured the total testing time to cover all possible execution paths.

5.3 Experimental Results

Regarding RQ1: How does the distributed concolic algorithm improve the speed of concolic testing the MSR code. We performed 4 series of experiments with 4 to 5 PUs with 5 to 6 LSes and with 1, 4, 8, 12, and 16 computing nodes. The total numbers of test cases generated and corresponding time costs are reported in Table 2. For example, 1.1×10^5 test cases were generated for MSR with 4 PUs w/ 5 LSes (see the last column of Table 2). 1 computing node took 643 seconds to generate 1.1×10^5 test cases for the experiment with 4 PUs and 5 LSes. However, 4, 8, 12, and 16 nodes took only 186, 89, 60, and 45 seconds for the same experiment, respectively (see the second row of

Table 2. Total number of generated test cases and time costs (seconds)

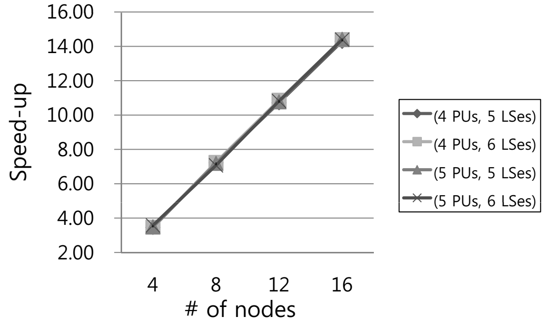
	1	4	8	12	16	# test cases
4 PUs w/ 5 LSes	643	186	89	60	45	1.1×10^5
4 PUs w/ 6 LSes	3194	919	441	294	222	5.3×10^5
5 PUs w/ 5 LSes	3242	927	451	301	225	4.9×10^5
5 PUs w/ 6 LSes	19225	5369	2718	1777	1336	2.8×10^6

Table 2). Therefore, compared to the original concolic testing (the second column of Table 2), the distributed concolic testing reduced time cost significantly.

In a similar manner, we can analyze speedup achieved by the distributed concolic algorithm. Figure 5 illustrates speedup results with a different number of computing nodes. For example, with an environment containing 5 PUs and 6 LSes (the last row of the table in Figure 5), 4 computing nodes completed all testing 3.58 times faster than 1 computing node ($3.58 = \frac{19225}{5369}$). Similarly, 8, 12, and 16 computing nodes with the same environment completed concolic testing 7.07, 10.82, 14.39 times faster, respectively.

Node #	4	8	12	16
4 PUs w/ 5 LSes	3.46	7.22	10.72	14.29
4 PUs w/ 6 LSes	3.48	7.24	10.86	14.39
5 PUs w/ 5 LSes	3.50	7.19	10.77	14.41
5 PUs w/ 6 LSes	3.58	7.07	10.82	14.39

(a) Table of speed-up ratios for different numbers of nodes



(b) Graph of speed-up ratios for different numbers of nodes

Fig. 5. Speed-up ratios for different numbers of nodes

Regarding the effectiveness of bug detection, the results of the distributed concolic algorithm are the same as the ones of the original concolic testing. The distributed concolic testing detected violations of the requirement property (`assert(` $\forall i.$`LS[`i`]==buf[`i`])`) due to the all 9 bugs in a few seconds. Thus, the distributed concolic testing reduces time costs significantly without loss of effectiveness compared to the original concolic testing.

Regarding RQ2:How does the distributed concolic algorithm achieve scalability when applied to the MSR code. As shown in the table of Figure 5, the efficiency of parallelism ($\frac{\text{speedup}}{\text{\# of nodes}}$) is almost 90% regardless of the number of nodes. For example, MSR with an environment consisting of 5 PUs and 6 LSes showed almost the same parallelism efficiency for different numbers of nodes (see the last row of the table in Figure 5 where $\frac{3.58}{4} \approx \frac{7.07}{8} \approx \frac{10.82}{12} \approx \frac{14.39}{16} \approx 90\%$). Furthermore, as shown in the graph of Figure 5, the distributed concolic algorithm achieved almost identical parallelism efficiency regardless of the environment configurations. In other words, 4 PUs with 5 LSes, 4 PUs with 6 LSes, 5 PUs with 5 LSes, and 5 PUs with 6 LSes achieve almost identical parallelism efficiency. This observation indicates that the performance of the distributed concolic algorithm is *not* affected by the complexity of a target program either, which is another advantage of our distributed concolic algorithm.

Table 3. Overhead due to the distributed concolic testing

	Waiting time (%)				Communication (%)				Waiting + Communication (%)			
	4	8	12	16	4	8	12	16	4	8	12	16
4 PUs w/ 5 LSes	2.54	2.63	2.65	2.79	1.09	1.22	1.84	1.72	3.64	3.85	4.49	4.51
4 PUs w/ 6 LSes	2.46	2.56	2.58	2.61	1.31	1.42	1.45	1.78	3.77	3.98	4.03	4.39
5 PUs w/ 5 LSes	2.23	2.28	2.30	2.33	1.49	1.56	1.53	1.21	3.72	3.85	3.83	3.54
5 PUs w/ 6 LSes	2.19	2.11	2.16	2.17	1.15	3.55	1.50	1.46	3.35	5.67	3.66	3.64

Table 4. Numbers of iterations performed by nodes

# nodes	(PUs, LSes)	Node ID																Avg	Stdev
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16		
4	(4,5)	27.4	31.5	27.0	25.3													27.8	2.6
($\times 10^3$)	(4,6)	122.6	136.6	124.2	151.4													133.7	13.4
	(5,5)	117.3	121.1	134.8	116.0													122.3	8.6
	(5,6)	771.0	690.0	645.4	670.7													694.3	54.3
8	(4,5)	13.4	14.0	13.0	13.6	14.6	13.5	15.1	14.0									14.0	0.7
($\times 10^3$)	(4,6)	63.5	63.8	63.6	69.6	67.6	62.6	71.5	72.6									66.9	4.0
	(5,5)	59.8	68.9	58.5	59.0	59.3	60.4	62.3	60.9									61.1	3.4
	(5,6)	335.3	334.5	355.4	332.7	399.1	347.0	332.9	340.2									347.1	22.5
12	(4,5)	8.9	9.2	10.4	8.9	9.2	8.6	9.0	8.8	9.7	10.5	9.2	9.0					9.3	0.6
($\times 10^3$)	(4,6)	42.7	42.9	48.4	42.3	49.3	42.8	45.5	43.0	42.4	41.5	44.2	49.7					44.6	3.0
	(5,5)	42.9	39.2	40.8	38.7	40.1	39.3	42.4	40.3	39.5	40.3	39.6	46.2					40.8	2.1
	(5,6)	216.2	239.0	219.8	221.6	220.7	239.8	233.0	249.8	272.9	222.0	222.6	219.5					231.4	16.7
16	(4,5)	7.2	7.3	7.4	6.9	7.0	7.2	6.8	6.6	6.3	6.6	6.6	7.4	7.2	7.7	6.4	6.8	7.0	0.4
($\times 10^3$)	(4,6)	32.2	31.8	34.0	32.7	34.1	33.4	32.2	37.3	31.3	37.3	32.6	32.1	32.7	32.1	32.5	36.5	33.4	2.0
	(5,5)	30.1	29.4	31.8	29.9	32.1	32.4	29.4	33.3	29.7	29.1	31.6	28.9	29.1	30.4	30.9	31.2	30.6	1.4
	(5,6)	167.2	207.5	167.3	189.6	168.5	167.1	166.5	167.9	170.5	176.0	174.4	174.0	165.3	180.3	171.0	163.9	173.6	11.2

We expect that a high parallelism efficiency can be achieved even with a large number of computing nodes (saying hundreds of thousand computing nodes of cloud computing), since there is little dependency among computing nodes. Communication between two nodes occurs only when one of the nodes has completely generated and analyzed all possible subsequent symbolic paths from a given symbolic path (i.e., when $queue_{pf}$ is empty). Thus, each node can concentrate on its own computational task with little waiting/blocking for other nodes. For the similar reason, communication costs caused by the distributed concolic algorithm are also insignificant.

Table 3 describes overhead caused by the distributed concolic algorithm. For example, with 4 computing nodes, MSR with an environment model of 4 PUs and 5 LSes spent 2.54% of the whole execution time (on average) by waiting/idling until it received a symbolic path formula from another node (between line 27 and line 28 of Algorithm 2). In addition, MSR with the same environment spent 1.09% of the whole execution time for socket communication. Thus, these two overheads of the distributed concolic algorithm constitute 3.64% of the whole execution time. The remaining 6% ($\approx 10\% - 3.64\%$) of overhead is caused by the increased complexity of the distributed concolic algorithm such as maintaining $queue_{pf}$ and handling communication at user process level, etc. Considering that the current implementation of the distributed concolic algorithm is not optimized, this overhead can be reduced further.

Another evidence of the scalability of the algorithm can be found in Table 4, which describes the numbers of iterations (test case generations) performed by the computing

nodes. For example, with MSR with an environment model of 4 PUs and 5 LSes, nodes 1, 2, 3, and 4 performed 27.4×10^3, 31.5×10^3, 27.0×10^3, and 25.3×10^3 iterations, respectively. The numbers of iterations for different nodes are roughly similar, which means that an equal amount of work was assigned to each node, which improves global utilization of computing nodes. Note that time cost for each iteration may vary depending on the length of a corresponding symbolic path formula. Measured time costs of nodes (not shown in this paper) are even more identical.

6 Conclusion and Future Work

We have developed a distributed concolic algorithm which can reduce time cost by utilizing a large number of computing nodes. Furthermore, we have demonstrated the improved performance of the algorithm through an industrial case study on the multisector-read operation of a Samsung flash storage platform. We applied the distributed concolic algorithm to the MSR code and analyzed the approach empirically. In this case study, the distributed concolic algorithm achieved an order of magnitude faster testing speed compared to the original concolic algorithm while maintaining the effectiveness of bug detection capability. Although these experiments were performed on only 16 computing nodes, we could observe that the algorithm has good characteristics of scalable distributed algorithms such as linear speedup with an increasing number of nodes, little blocking time, and nominal communication overheads. Therefore, we expect that the distributed algorithm can alleviate problems caused by heavy computational costs in a large degree.

We plan to apply the distributed concolic algorithm to target programs on 10,000 nodes of the Amazon EC2 platform [6] to demonstrate the scalability of the algorithm in a concrete manner. Furthermore, this experiment can suggest a promising direction of fighting the state space explosion problem of automated verification techniques. Finally, we will develop a new concolic algorithm for branch coverage for more practical applications in an industrial setting.

References

1. Andrews, J.H., Briand, L.C., Labiche, Y.: Is mutation an appropriate tool for testing experiments? In: International Conference on Software Engineering, ICSE (2005)
2. Burnim, J.: CREST - automatic test generation tool for C, `http://code.google.com/p/crest/`
3. Burnim, J., Sen, K.: Heuristics for scalable dynamic test generation. Technical Report UCB/EECS-2008-123, EECS Department, University of California, Berkeley (September 2008)
4. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Operating System Design and Implementation, OSDI (2008)
5. Dutertre, B., Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
6. Amazon Elastic Compute Cloud (Amazon EC2), `http://aws.amazon.com/ec2/`

7. Ferreira, M.A., Silva, S.S., Oliveira, J.N.: Verifying Intel flash file system core specification. In: 4th VDM-Overture Workshop (2008)
8. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: Programming Language Design and Implementation, PLDI (2005)
9. Godefroid, P., Levin, M.Y., Molnar, D.: Automated whitebox fuzz testing. In: Network and Distributed Systems Security (2008)
10. Kang, E., Jackson, D.: Formal modeling and analysis of a flash filesystem in Alloy. In: Abstract state machines, B and Z (2008)
11. Kim, M., Choi, Y., Kim, Y., Kim, H.: Formal verification of a flash memory device driver - an experience report. In: Spin Workshop (2008)
12. Kim, M., Kim, Y.: Concolic testing of the multi-sector read operation for flash memory file system. In: Oliveira, M.V.M., Woodcock, J. (eds.) SBMF 2009. LNCS, vol. 5902, pp. 251–265. Springer, Heidelberg (2009)
13. Kim, M., Kim, Y., Kim, H.: Unit testing of flash memory device driver through a SAT-based model checker. In: Automated Software Engineering (ASE) (September 2008)
14. King, J.C.: Symbolic execution and program testing. Communications of the ACM 19(7) (1976)
15. Samsung OneNAND fusion memory, http://www.samsung.com/global/business/semiconductor/products/fusionmemory/Products_OneNAND.html
16. Sen, K., Marinov, D., Agha, G.: CUTE: A concolic unit testing engine for C. In: European Software Engineering Conference/Foundations of Software Engineering, ESEC/FSE (2005)
17. SMT-LIB: The satisfiability module theories library, http://combination.cs.uiowa.edu/smtlib/
18. Tillmann, N., Schulte, W.: Parameterized unit tests. In: European Software Engineering Conference/Foundations of Software Engineering, ESEC/FSE (2005)
19. Visser, W., Pasareanu, C.S., Khurshid, S.: Test input generation with Java PathFinder. In: International Symposium on Software Testing and Analysis, ISSTA (2004)